2016 年度　修士論文

開発委託先変更に対するソフトウェアの複雑さ・不具合修正回数に関する研究

2017 年 1 月 30 日（月）提出

指導：深澤　良彰 教授

早稲田大学　基幹理工学研究科
情報通信・情報理工学専攻　深澤研究室

学籍番号：5115F002-0
阿部　晃佑

目次

1. はじめに

ソフトウェアの開発規模は、年々大規模になってきている。これに伴って、時間的にも人員的にも多くのコストがかかってしまう。また、開発プロセスの上流工程にあたる要求定義や設計は開発プロセスの中でも特に重要な工程であり、これらに多くのコストを割きたい。しかし、開発規模の大規模化と相まって、コスト不足から重要な工程にコストを割くことができない。

これを解決するために、開発プロジェクトの一部の実装を外部企業に委託している企業も少なくない。この手段を用いることによって、開発コストを削減したうえで大きな利益を得られることを目的としている。一方で、開発を外部に委託する際に発生する問題もある。それは外部が実装したソフトウェアの品質が望ましいものでない可能性があるということである。

ソフトウェアの品質とは、そのソフトウェアの設計がきれいでわかりやすいものになっているか、変更を加えたときに修正する箇所が多すぎないか、今後再利用する際に問題はないかなどといった完成度に関する指標である。この品質は、ソフトウェアを様々な尺度から測定することで得られる値を用いて評価されることが多い。

このソフトウェアを測定するための様々な尺度をメトリクスという。測定する対象は多岐におよび、ソースコードの変更回数といった工程に関するもの、コードの行数といった開発物に関するもの、開発に関わった技術者の人数といったリソースに関するものの大きく３つに分けられる。

外部が実装を行うことで、一旦は時間的にも人員的にもコストを削減することはできる。しかし、開発物の品質が望ましいものでなかった場合には、一度納品を受け入れた後に直すべき不具合が多く見つかり、外部委託先へ開発物の手戻りが何度も発生してしまう。手戻り回数が増加は納期の遅れへ直接関係するため、結果的には削減したと思われた時間的コストが削減できていなかったということになりかねない。

また、外部によって実装された開発物を基にして機能の追加や変更といった派生開発を行うとしたときに、ソフトウェアの変更や修正が行いやすいかどうかを見る品質特性を保守性と呼ぶ。この保守性が望ましい品質でなかった場合には、修正すべき箇所が必要以上に多くなって不具合を引き起こしやすくなったり、あまりにも複雑な実装のため派生開発そのものが不可能であったりするケースも考えられる。

この問題を解決するためには、開発を行う外部企業にとってソフトウェア品質指標となるものが必要であると私は考えた。この品質指標においてある一定

の基準を満たしたソフトウェアでないと受け入れを認めないとすれば、一定の基準を満たしたソフトウェアのみを受け入れることが可能となる。

このような指標を定めるにあたって、何が必要となるかを考えたとき、ソフトウェアの品質と直接関係のある、ソースコードのメトリクスが挙がった。

本論文では、共同研究を行っている建機メーカー（以下共同研究先を表記する）が過去に外部委託をして開発されたプロダクトを用いて、外部委託開発されたソフトウェアの品質指標を定めるのに重要となるメトリクスの測定値にどのようなものがあるかを調査する。調査の際には、２つのメトリクス間の散布図や相関係数、ファイル単位で測定されるメトリクスの分布図の観察を行った。

また、それらのプロダクトを受け入れた後にどのような不具合が起きたかを調査し、メトリクスの選定に有益となる特徴がないかを調査した。こちらの調査では、実プロダクトの不具合とその原因が自由言語で書かれた文書を筆者が人手で原因の分類を行った。

更に、そのプロダクトのうちいくつかは開発途中でソフトウェア開発元が変更されたものがあった。我々はそれらのプロダクトの委託先変更前時点でのプロダクトメトリクス値（以下、元測定値）と委託先変更後のプロダクトメトリクス値の変化量(以下、前後差分値)と不具合修正回数の関係について着目した。

本研究課題を以下に定義する。

RQ1 外部委託開発をしたソフトウェアに見られるメトリクス測定値の共通の傾向はあるか

RQ2 一度納品された後に検出された不具合の原因について、複数のプロジェクトでどのような共通点があるか

RQ3 元測定値と前後差分値の関係は、開発委託先変更の有無によって違いがあるか？

RQ4 元測定値、前後差分値と不具合修正回数の間にはどのような関係があるか？

RQ5 元測定値、前後差分値と不具合修正回数の間にある関係は、開発委託先変更の有無によって違いがあるか？

以上の５つの研究課題を明らかにするための調査を行い、調査結果について考察を行った。調査対象は、共同研究先で実際に外部に開発を委託したソフトウェアの合計１１プロジェクトである。

本研究の貢献を以下に示す。

・外部委託開発をしたソフトウェアにおける共通の品質特徴を確認した。

・外部委託開発をしたソフトウェアにおける共通の不具合原因傾向を確認した。

・委託先変更を伴うソフトウェア開発において、元測定値が品質基準を満たしていなかった場合に、Cyclomatic 複雑度の前後差分値が大きくなる危険性があることを確認した。

・委託先変更を伴わないソフトウェア開発において、元測定値が品質基準を満たしている場合でも Cyclomatic 複雑度の前後差分値が大きな値を示す結果が得られた。ただし、最終成果物としての品質は保たれているため、委託先変更を伴う場合に比べて品質低下のリスクが低いことを確認した。

・開発委託先変更の有無にかかわらず、前後差分値が大きい関数は不具合修正回数が多く、委託先変更の有無は不具合修正回数と関連がないことを確認した。

以下に本論文の構成を述べる。２章では本研究の背景について述べる。続く３章では提案手法について説明する。４章では調査方法とその結果、考察を述べる。５章では関連研究について説明し、その関連研究から本研究がどのように発展していく可能性があるかを述べる。６章で本論文のまとめについて述べる。

2. 背景

2.1. ソフトウェア開発の現状

ソフトウェア開発は、大きく「要求定義」、「設計」、「実装」、「テスト」の 4 工程に分けることができる。「要求定義」や「設計」は、ソフトウェアの根幹を成す部分であり、これらの工程に大きくコストを割いて要求や設計の抜けや漏れがないようにすることが望ましい。これが実現できればその後の「実装」工程もスムーズに進み、高品質なソフトウェアが開発されると考えられる。

しかし、現状のソフトウェア開発ではこれらの工程にコストを多く割くことが難しい。原因は、ソフトウェア開発規模の大規模化によるものである。開発規模の大規模化に伴って、開発全体にかかるコストも増大している。そのため、「要求定義」や「設計」といった工程に多くコストを割くことは非常に困難である。

もし実際に、この２つの工程にコストを無理やり多く割いた場合にはどうなるだろうか。「要求定義」や「設計」の工程に大きく時間的コストを割くためには、開発期間を延ばす手段と、その後の「実装」と「テスト」工程にかける時間を短くするかの 2 つが考えられる。後者の方法では、本来費やせるはずだった時間が短くなるため、「実装」が期日内に間に合わない場合や、「テスト」の漏れがあって見つけるべき不具合が後々になってから見つかり余計な時間がかかるといったケースを引き起こす可能性が上がってしまう。かといって、前者の手段を用いた場合には、開発期間が延びることで多くの人員がその期間に拘束されることになり、時間的コストと人員的コストの両方を多く割くことになってしまう。

このような現状から、開発全体のコストを削減しながら品質に問題が生じない開発方法が求められる。

2.2 開発の外部委託

前述の問題を解決するために、「実装」工程のほとんどと「テスト」工程の一部を外部企業に委託しているケースが多くみられる。本項では、外部委託開発においてどのような種類があり、具体的な開発プロセスのうちどのプロセスを外部に委託するのかを説明する。また、外部委託開発において生じる問題についても触れる。

2.2.1 外部委託の種類

一言に外部企業といってもいくつかの分類分けがある。それについてまず説明する。

分類をする基準の一つ目は、資本的関係の有無である。ここでいう資本的関係とは、外注先が子会社や関連会社であるかどうかである。資本関係のない企業に開発を委託する場合には「アウトソーシング」、資本関係のある企業である場合には「インソーシング」と分類される。二つ目の基準は、地理的関係である。具体的には国内企業であるかと海外企業であるかで分類される。海外企業に依頼する場合には「オフショア」、国内企業である場合には「オンショア」や「ニアショア」と呼ばれる。これらの関係についてまとめたものを以下の表 1 に示す。本研究では、外部企業へ実際に開発を委託したプロダクトを分析対象と用いる。これらは国内の資本関係のない企業へ開発を委託されたものであるので、「ニアショアアウトソーシング」にあたる。これは、表の中の太線で囲まれた部分である。

	資本関係の有無	
	有り	無し
国内	オンショア インソーシング	オンショア アウトソーシング
国外	オフショア インソーシング	オフショア アウトソーシング

表 1 外部委託開発の種類

2.2.2. 具体的な開発プロセス

開発プロセスにおける「実装」と「テスト」の一部を外部に委託すると前述した。その開発プロセスについて具体的な図を用いて説明する。ソフトウェアの開発プロセスは、しばしば V 字型のモデルで表現される。それが以下の図に示したものである。この図では、赤枠で囲んだ部分を外部に、黒枠で囲んだ部分を自社で行うように示している。これは、本研究で用いた実プロダクトにおける状況である。別の企業が外部に開発を委託した際には若干の違いがある可能性がある。

開発プロセスは「要求定義」、「設計」、「実装」と「テスト」の大きく４つに分類されることは前述した。この図では、「設計」プロセスが「外部設計」と「内部設計」の二つに分けられている。これらは、それぞれ以下のように定義している。

・外部設計…ソフトウェアのインタフェースに関する設計。クラスの名前やクラス同士の関係はこの外部設計でされる

・内部設計…クラス内部にある関数などをどのように実装するかを示したもの

図では、このうちの「外部設計」は自社で行い、「内部設計」は外部に委託するように示している。「外部設計」を自社で行っているのは、作成されるファイル、クラス、関数の名前を指定するためである。これらを指定しなかった場合、それぞれの名前を外部が任意につけることになってしまい、場合によっては内容の把握が全くできなくなってしまうためである。一方の「内部設計」は外部に委託している。この設計プロセスは「実装」のプロセスと深く関連するものであるため外部への委託を行っている。

次に「テスト」プロセスについて説明する。これは図中で「ユニットテスト」と「統合テスト」、「システムテスト」の３つに分かれている。それぞれのテストは、同じレベルに並んでいるそれぞれのプロセスが正しいものであったかを検証するものである。「内部設計」は外部委託先が行いっており、それが正しいものであるかを検証するプロセスとして、「ユニットテスト」の工程を外部に委託している。共同研究先では、「ユニットテスト」プロセスにおけるテストケースも自社で用意はしているが、より多くの不具合を発見するために、外部が行う「ユニットテスト」と自社で行う「ユニットテスト」の２重でテストを行っている。

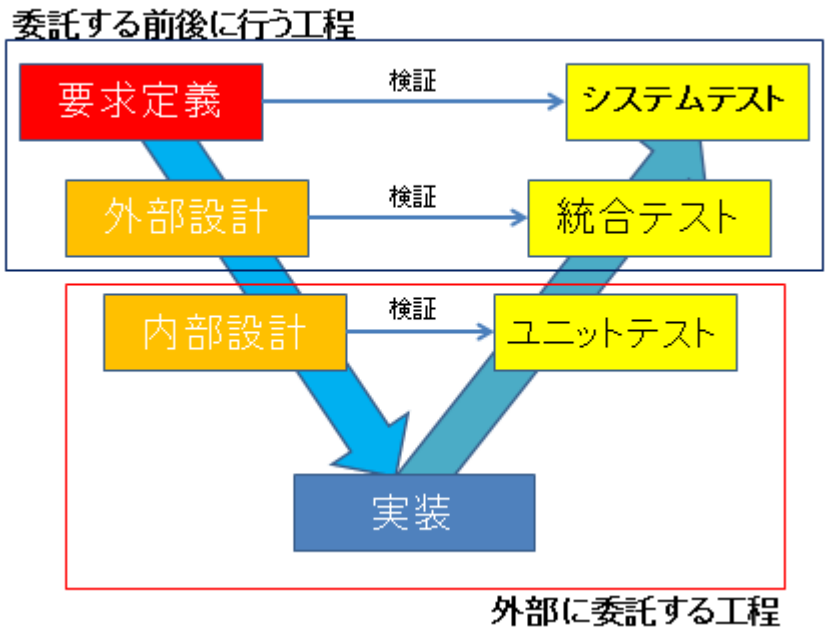

図１　Ｖ字型開発プロセス

2.3 外部委託開発における問題点

一方でこのソフトウェアの外部委託開発にも問題点がある。まず挙げられるのは、外部企業が納品してきた開発物に多くの不具合が残存していた場合である。外部委託開発の場合、納品後に新たな不具合が発見された場合には一度開発物を委託先へ戻して修正を行ってもらう手戻り作業が必要となる。初めに納品された開発物に多くの不具合が残存していた場合には、手戻りを何度も繰り返して不具合を徐々に改善していくため当初予定していた期間よりも開発が長引く恐れがある。

また不具合の問題とは別に、ソフトウェアの品質が悪かった場合に生じる問題がある。この品質において、「保守性」が低いソフトウェアを外部企業から受け入れた場合を考える。機能要求は満たしており、受け入れたソフトウェアをそのまま利用する場合にはとりあえずは問題がない。しかし、このソフトウェアに新たな機能を追加したいといった状況になった場合には問題である。なぜなら「保守性」が低いということは、変更を加えることが困難であることを示すからである。追加機能を実装する際に、一つの変更を加えると同時に多くの変更を加えなければならない場合が想定される。これは不具合を引き起こす原因にもなり得るため、機能追加に多くの時間を費やすことが想定される。

受け入れたソフトウェアが「保守性」の高いものであったならば、機能追加をする際にもスムーズに作業が進み無駄な修正を加える必要もなくなるのである。「実装」工程を外部に委託したことによってその時には時間的コストを削減できるかもしれないが、今後そのソフトウェアを使い続けていく場合には品質を考慮しないと余計な時間がかかってしまうかもしれない。これはすなわち削減できたと思われた時間的コストが別のところにかかっていることになる。これでは外部委託開発における当初の目標であったコスト削減が実現していないことになってしまう。

2.4 開発委託先の変更

開発組織変更がソフトウェア品質に影響を与えるかどうかについては、既存研究でいくつか言及されている。本稿では、委託先変更は開発組織変更の一部であるとしたうえで、既存研究をふまえて我々の研究の新規性について述べる。

Seiji は独自に origin という委託先変更の変遷を示すメトリクスを定義して、ファイルを分類した。そのうえで、origin がプロダクトメトリクスやファイルの修正回数、欠陥数との関連を調査している。調査の結果として、委託先変更はプロダクトメトリクスをより複雑にしてしまうこと、コード修正回数を増加

させる傾向にあることを述べている。我々の研究では、「委託先変更」がソフトウェアに与える影響に加えて「元測定値」と「前後差分値」に着目している。

3. 提案

3.1 外部開発ソフトウェアの受入指標

前述した問題を解決するためには、外部開発ソフトウェアを定性的に評価して受け入れるかどうかを決定する「指標」が必要ではないかと私は考えた。ある開発物について、その「指標」を用いて評価をする。その評価がある一定の基準を満たしているのであれば受入完了とする。このようにすれば、受入完了となった外部開発ソフトウェアは品質の保証がなされており、受入完了後に発見される不具合件数や今後派生開発を行ったときの開発物における理解しにくいコードや不具合発生を大きく減らすことができるだろう。

外部委託開発の工程において、この「指標」をどの段階で利用すれば問題の解決につながるかを考える。以下の図が外部委託開発プロセスの大まかな流れである。

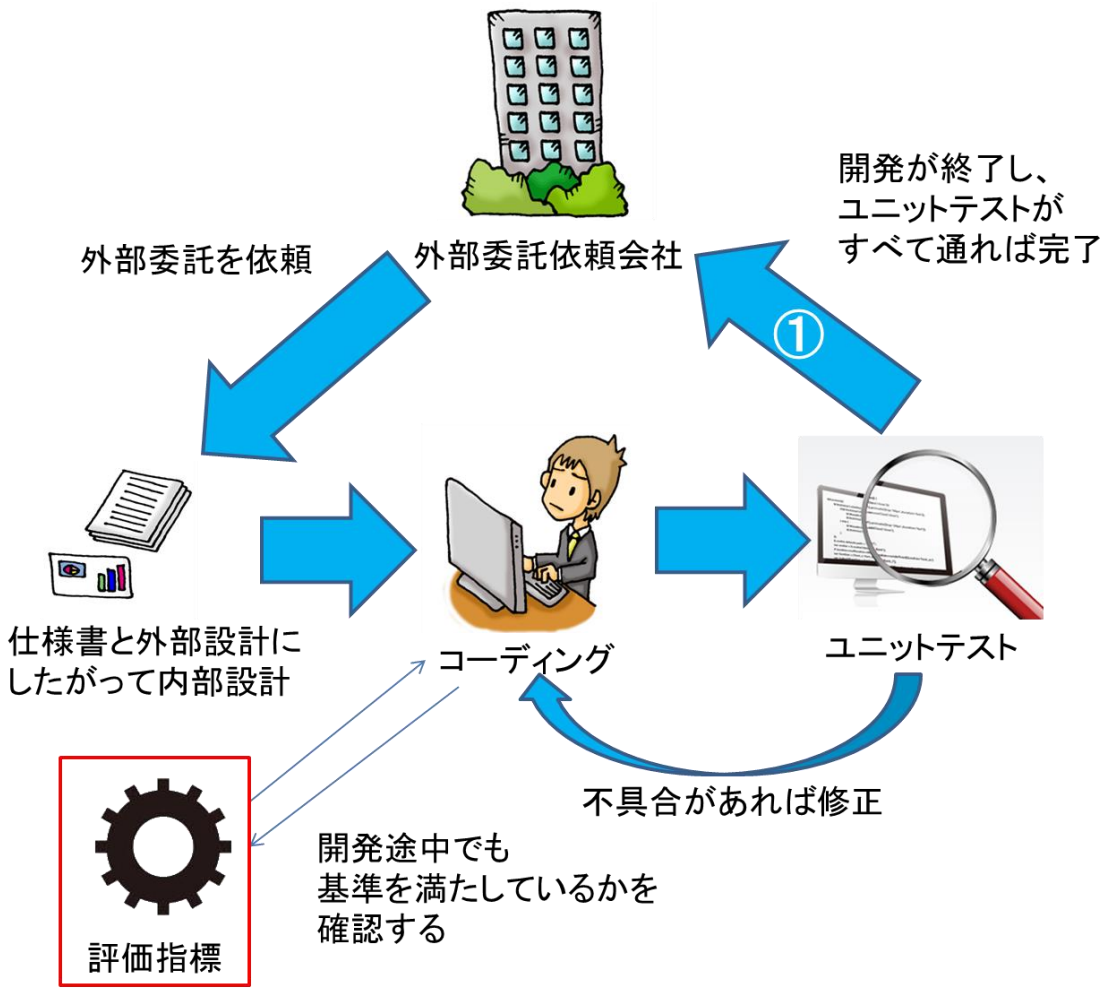

図 2　提案する外部委託開発のプロセス

これは、図１で示した「内部設計」、「実装」と「ユニットテスト」のプロセスについてより詳細に示したものである。またこの図では、今回提案した「指標」をどの段階で利用するべきかを示している。

ここで一つ考えなければならないことは、この基準を「いつ使うか」である。図２では、「実装」プロセス中に評価指標を使い開発途中に何度も評価を行うべきであると示している。「実装」プロセス中に評価を行うべきであると考えたのはなぜかを説明する。

外部開発ソフトウェアは、指定された期限までに納品される必要がある。ここで仮に、外部企業が開発を完了した段階である図２中の「①」と示された矢印のときに受入基準評価を用いたとする。受け入れてよいという結果が出ればそのまま納品となるため問題はないが、もし受け入れできないという結果になった場合は開発物に修正を加えることになってしまう。外部委託開発先が、開発が完了であると判断した時期が納品すべき期限の直前であったとしたら、受入拒否判定が出た後の修正期間によって、納期が守られなくなる恐れがある。このような事態を避けるためには、開発途中のソフトウェアであっても評価することができる必要がある。したがって、図２中のプロセスにおいて「実装」プロセス中に評価を行うべきであると示した。常に受入の基準を満たすように逐一評価をしながら開発を進めていくことによって、期日内に品質の保証されたソフトウェアが納品される。このようなプロセスが開発において望ましいものであるといえる。

3.2 基準を設定するために用いるデータ

本項では、基準を設定するにあたって分析を行うデータについて提案をする。

3.2.1 メトリクス

この基準を用いて外部開発ソフトウェアの品質を評価するためには、開発途中であっても得られるデータから評価を行う必要がある。そこで、まず分析をする対象として開発物のソースコードを選んだ。評価すべき対象そのものであり、実装の工程で確実に得られるデータだからである。このソースコードを静的解析ツールにかけることによって、定性的なデータであるプロダクトメトリクスの測定値を得ることができる。参考文献「metrics_2nd.pdf」

メトリクスとは、ある対象の定性的なデータを測定する様々な尺度である。メトリクスは大きく３つの種類に分けられる。それをそれぞれプロダクトメトリクス、プロセスメトリクス、リソースメトリクスと呼ぶ。これらの違いは、

何を対象に測定するかである。プロダクトメトリクスは、ソフトウェア開発における成果物であるプロダクトについて測定を行うもの。プロセスメトリクスは、ソフトウェア開発工程における各工程での活動について測定を行うもの。リソースメトリクスは、ソフトウェア開発作業において消費される時間や人手コストといったリソースについて測定を行うものである。

本研究において、多くのプロセスメトリクスとリソースメトリクスを測定するには非常に困難である。なぜならば、「外部委託開発をしたソフトウェア」に対象を絞っているためである。

まずプロセスメトリクスについて考える。プロセスメトリクスの例を挙げると、対象となるソフトウェアの「ソースコードの変更回数・変更率」、「コードレビュー実施の有無」や「欠損除去率」といったものがある。ソフトウェアの「実装」工程を外部に委託しているため、その工程における詳細な情報を得るには外部委託先との多くの情報交換が必要となる。更には、外部委託先が「ソースコードの変更回数・変更率」などを測るためのログデータを残しているとも限らないし、残していたとしてそれを提供してくれるという保証もない。

一部のプロセスメトリクスとして得られるものとしては、受入を一度認めた後に生じた「不具合件数」が挙げられる。これは外部委託開発が完全に終了した後に得られるデータではあるが、この「不具合件数」を開発途中のプロダクトメトリクスの関係性を探ることは有用な情報を得られる可能性がある。何故ならば、開発途中のプロダクトメトリクスから、「不具合件数」を予測するなどといったことが可能になる可能性があるためである。

次にリソースメトリクスについて考える。リソースメトリクスの主な例として挙げられるのは、「開発要員数」、「開発要員の経験年数」、「開発環境のスペック」などがある。これらについてプロセスメトリクスと同様に、それを測定しているとも限らなければ、測定したデータを提供してくれるという保証はないのである。

以上のことから、測定するメトリクスをプロダクトメトリクスと一部のプロセスメトリクスである「不具合件数」に絞り込んだ。プロダクトメトリクスでは、ファイルの実行コード行数やコメント行数が何行であるか、内部に記述されている関数の複雑度はどのくらいであるか、他ファイルとの呼び出し関係はどのようになっているかなど様々な視点からソースコードを測り数値を算出することができる。測り方は共通であるため、異なるファイルのメトリクス結果をそれぞれ算出して比較することでどのような差があるかを数値で見ることができる。

3.2.2 受入後に検出された不具合の原因分類

次に着目したのは、外部委託開発をしたソフトウェアを一度受け入れた後に生じた不具合についてである。この不具合は、実装を行った外部企業が検出できなかった不具合にあたる。これが多いプロダクトは、手戻りの回数が多くなり開発完了が遅れる。そのため、このような不具合は少ないことが望ましいことは前述したとおりである。その件数に加えて、「その不具合は何が原因で起こったのか？」を分析することが重要ではないかと考えた。この原因を分析することによって、検出されやすい不具合の傾向や開発のどの段階で埋め込まれたものであるのかを意味づけて考えることができる。またこの意味づけした結果と共に、前述したソースコードメトリクスと関連付けることによって、特に注目するべきメトリクスに目星をつけられるのではないかというのが狙いである。重要なメトリクスに目星をつけることによって、それを考慮したものを受け入れ基準設定の評価式に組み込める。

本研究では、実際に外部委託開発を行ったプロダクトのデータからこの傾向を得ることを想定している。プロダクトの詳細は 4 章の事例とデータで述べることとし、ここでは分類手法の説明をする。これは（参考文献 IPA のもの）を利用している。分類の一覧を表 3-1,3-2 に示す。

まず大きな分類として「工程」がある。これは参考文献の言葉を引用すると

引用“バグが発生した工程。ESPR におけるソフトウェア・エンジニアリング・プロセスに基づいて「４つの工程」と「工程共通」に分けています。”

とある。参考文献の元々の分類では、「工程」に「要求定義（外部）」という項目があったが、今回用いる分類分けでは要求定義を開発依頼社が行うことが前提であるためこの項目を排除した。「種別」がバグの分類をあらわし、「説明」の」項目で分類における説明が述べられている。

工程	種別	説明
要求定義	記述誤り	ソフトウェア要求仕様書などにおける記述で、要求されている機能全体の抜けによるもの。
	機能の欠如	ソフトウェア要求仕様書などにおける記述で、要求されている機能全体の抜けによるもの。
	機能の定義誤り	ソフトウェア要求仕様書などにおける要求の定義が誤っているもの。要求されていない機能が追加されているものも含む。
設計	データの誤り	データの取り扱いに関する誤りによるもの。
	アルゴリズム/制御の誤り	計算手順や演算方法に関する誤りによるもの。
	インタフェースの誤り	インターフェース仕様（設計）関係の誤りによるもの。 ・システム間のデータ形式（構造、量）の誤り。 ・プログラム、タスク間のデータ形式の誤り。 など その他、無用な依存関係を持ったインターフェースもここに分類する
	タイミングの誤り	タスク間のタイミング関係の誤り、設計不十分によるもの。 ・タスク間の実行条件（処理順序や割り込み処理の優先順位）の誤り。
	リソースの誤り	リソースの確保・解放忘れによるもの。 ・処理負荷の見積もりに関するもの。
	エラーチェックの誤り	エラーチェックの抜けによるもの。 ・関数、メソッド呼び出しの戻り値の扱いの誤り（エラーチェック抜けなど）。 ・入力データのチェックの誤りなど。
	記述の誤り	設計書における上記種別以外の記述の間違い、不明瞭、漏れなどによるもの。
	機能の欠如	設計書における記述で、要求されている機能全体の抜けによるもの。
	機能設計の誤り	設計書における機能の設計全体が誤っているもの。要求されていない機能が追加されているものも含む。

表 2-1　不具合原因分類表（要求定義と設計工程）

実装	データの誤り	コードレベルでのデータの取り扱いの誤りによるもの。
	ロジックの誤り	コードレベルでのロジック関係の誤りによるもの。 ・ブランチ：飛び先誤り。条件判定誤り。判断の抜け。判断内容誤り。 ・ループ：終了条件の誤り、ループ回数誤り。初期設定誤り。 ・四則演算処理誤り。 ・不要ロジックあり。ロジック抜け。ロジック位置不適当など。
	インタフェースの誤り	コードレベルでのインターフェース関係の誤りによるもの。 ・関数・メソッド呼び出しの引数の誤り。 ・他社製ソフトウェア（購入品）の設定や呼び出し誤り。
	タイミングの誤り	コードレベルでのタスク間のタイミング関係の誤りによるもの。 ・タスク間の実行条件（処理順序や割り込み処理の優先順位）の誤り。
	リソースの誤り	コードレベルでのリソースの確保・解放に関する誤りによるもの。
	エラーチェックの誤り	コードレベルでのエラーチェックの抜けによるもの
	機能の欠如	コードの記述で、要求されている機能全体の抜けによるもの。
	機能の実装誤り	上記以外で機能の実装が正しくないもの。要求されていない機能に対するコードが追加されているものも含む。
工程共通	統合の誤り	モジュール統合時の間違いによるもの。
	データベースの誤り	データベース利用方法の誤りによるもの。
	OS/パッケージソフトウェアの誤り	他社製ソフトウェア（購入品）に関する問題（ソフトウェアの動作不良など）によるもの。なお、他社製ソフトウェア組込み時の他社製ソフトウェアの設定や呼び出し誤りは、「コード」の「インタフェースの誤り」に分類する。
	その他	上記以外をこの種別で扱う。しかし、バグ分析を行う開発プロジェクトや組織の特性から、さらに種別の項目を追加し分類することもある。

表 2-2 不具合原因分類表（実装工程と工程共通）

3.2.3 開発委託先の変更前後の測定値

開発委託先の変更があったプロダクトは、委託先変更前のプロダクトを測定したメトリクスの値と変更後のプロダクトを測定したメトリクスの値に分類することができる。本研究では、開発委託先変更がソフトウェア品質に与える影響を調査するために、元測定値と前後差分値という二つの値を定義した。元測定値は開発委託先変更をする前に測定できるメトリクスであり、委託先変更後の変化量である前後差分値との関連を明らかにすることは、実際に委託先変更を伴うソフトウェア開発における変化量予測を可能とする。前後差分値を測定するには、元測定値の測定時点と開発最終時点において「同一の関数」が存在している必要がある。元測定値測定時点では存在していた関数が、最終開発時点では削除されていた場合、変化量を測定することはできない。同様に、元測定値測定時点では存在していなかった関数が開発最終時点までに新たに作られた場合も変化量は測定できない。そのため、本研究で調査対象とするのは「元測定値測定時点と開発最終時点において同一関数が存在しているもの」とする。

4. 調査

4.1 事例

本研究では、共同研究先が実際に外部に開発を委託したプロダクトを用いて分析を行った。国内の企業に組み込みソフトウェア開発の一部を委託している。以下にてそれらのプロダクトについて詳しく説明する。

4.1.1 データ

対象となるプロダクトは三つあり、それぞれが細かいプロダクト単位に分けられる。プロダクト A は、外部委託先にあたる A 社が開発したものである。これは三つの細かいプロダクトに分けられる。これをそれぞれ A1、A2、A3 と呼ぶことにする。プロダクト B は、細かく四つのプロダクトに分けることができる。これは細かいプロダクト単位で開発を委託した企業が異なる。細かいプロダクトをそれぞれ B1、B2、B3、B4 とすると、B1、B2 は B 社が開発を行い、B3、B4 は C 社が開発を行った。プロダクト C はプロダクト B と同様に 4 つの細かいプロダクトに分けられる。他の二つと同じように細かいプロダクトを C1、C2、C3、C4 とする。これの開発を行ったのはプロダクト B の一部を開発した C 社である。開発言語は A1 が C と C++、それ以外は C#となっている。これらの関係と開発規模について以下の表 3 に示す。

プロダクト大分類	プロダクト小分類	外部委託先	規模(LOC)	開発言語
A	A1	A社	15163/9741	C/C++
	A2		16700	C#
	A3		8576	C#
B	B1	B社	321	C#
	B2		1454	C#
	B3	C社	1167	C#
	B4		1133	C#
C	C1	C社	605	C#
	C2		3760	C#
	C3		2065	C#
	C4		1545	C#

表 3　プロダクト分類表

続いて、提案で述べた不具合原因の分類に用いたデータについて述べる。共同研究先では、外部委託開発を行う際には「不具合管理/問い合わせ台帳」とい

うものを用いて、不具合の管理や、開発過程で外部企業が疑問に思ったこと、逆に依頼側が疑問に思い問い合わせた内容を管理している。「不具合管理台帳」に記述される不具合内容は、一度依頼側が受入を行った後に検出された不具合に限る。受入後に不具合が検出された場合、この台帳にどのような原因が起こったのかを記述し、外部委託開発先へ修正を依頼する。それに伴って外部委託開発先で原因の究明と不具合の修正を行い、何が原因でその不具合が起こっていたのかとその修正方法を記述してもう一度プロダクトの納品を行うという形になる。問い合わせ台帳についても基本的な使用方法は同じであるが、こちらには単純な不明点を問い合わせるのに加えて、不具合と断定できないが怪しいと思われるものを「問い合わせ」という形でこちらの台帳に記入されることがある。怪しいと思われたものが、調査をした結果不具合であるとわかったということもあるためそのような案件も不具合とカウントする必要がある。

4.1.2 開発変更先の変更が有ったプロダクト

図 3 に本研究で用いた開発先変更のあったプロダクトの開発工程を示す。本研究において調査対象は、企業間の外部委託によって開発されたソフトウェアである。前述の通り、開発の工程は図 1 に示している。開発企業 A は、プロダクト A、B を外部企業 B、プロダクト C、D を外部企業 C に委託する形で開発を行った。初めの外部委託開発は図 1 の「First development process」の期間に行われた。本工程にて開発が一度完了した時点で企業 A は納品を受け入れた。その後、それぞれのプロダクトに機能追加を行うため「Second development」を外部に委託する必要があった。「Second development」を行っていく際には、４つのプロダクトを全て外部企業 B に委託する形で開発を行った。以上の開発工程を経たプロダクトを以下の 2 群に分けることで委託先変更の有無を区別する。

・委託先変更有（企業 C⇒企業 B）：プロダクト C, D
・委託先変更無（企業 B のまま）：プロダクト A, B

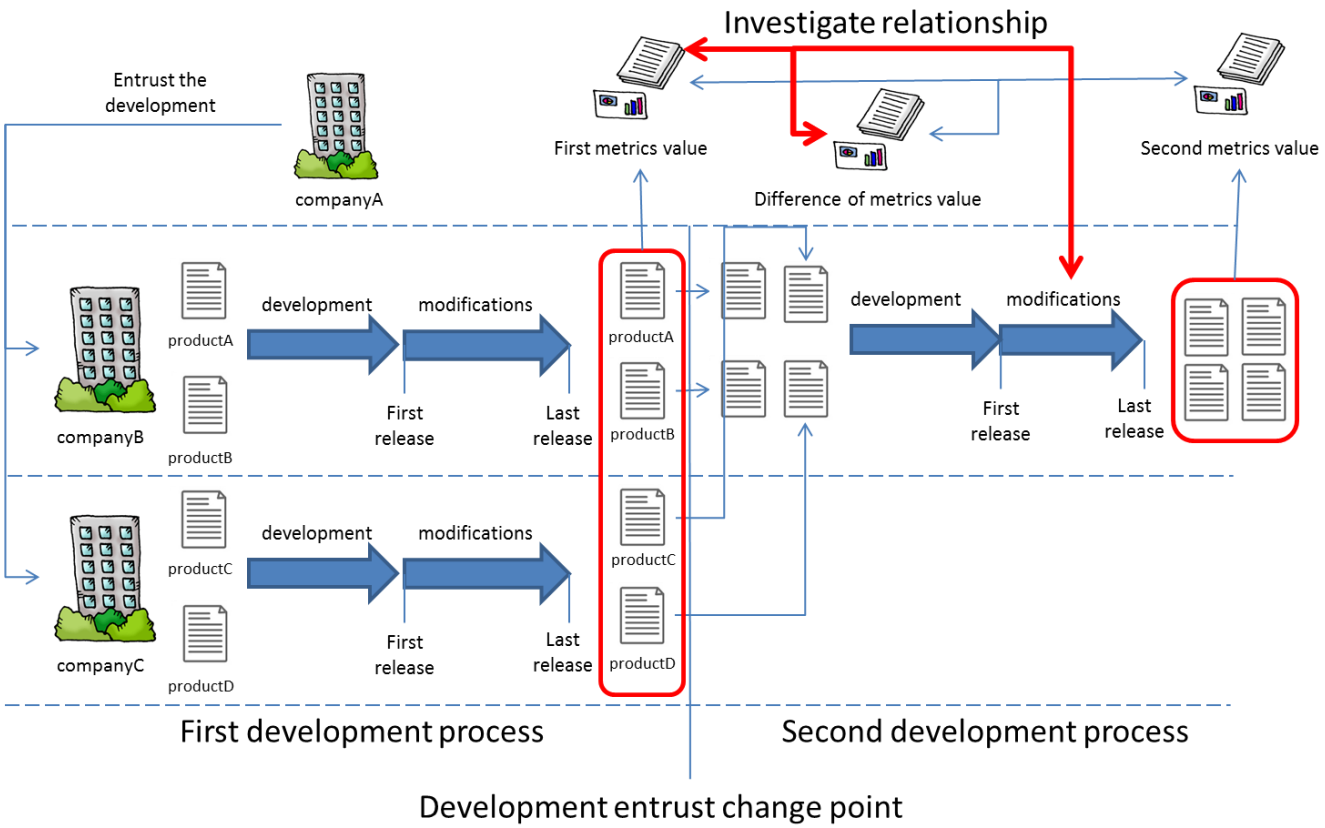

図 3　事例で用いる開発工程の概要図

4.2 分析方法

4.2.1. メトリクス

プロダクトメトリクスの測定には、Understand という静的解析ツールを用いた。このツールでは、一番大きな単位にあたるプロダクトそのものに加えて、それを構成するファイル、内部メソッドにおける様々なメトリクスを測定することができる。非常に細かい単位で多くのメトリクスを測定することができるのだが、今回はまず細かいプロダクトにおけるファイル単位のメトリクスを測定した。ここでいうファイルとは、C#のコードであれば拡張子が“.cs”、C++のコードであれば“.cpp”となるソースコードファイルを指す。ファイル単位で測れるメトリクスとしては主に以下のようなものがある。

・複雑度に関するもの
例）Cyclomatic 複雑度（平均、最大値、合計値）、Essential 複雑度（平均、合

計値）

ソースコードの複雑さを数値化したもの。複雑度は関数やメソッド単位で測られるため、ファイル内部に含まれる関数やメソッドそれぞれの複雑度を測り、そのうちの最大値であったり、すべての合計値であったり平均をとったものがファイル単位では得られる。複雑度は主に分岐数を計測するものだが、if 文や for 文による分岐だけでなく、case 文の分岐一つ一つを 1 と数えるか case 文一つで 1 と数えるかなど若干の違いがある。そのためいくつかの種類があり、用途に応じて使い分けることが必要になる。

・行数に関するもの

例）物理行数（空白行を含む)、実行コード行数（空白やコメント行を除いたもの)、コメント行数、セミコロン数、コード内のコメント行数率

ファイルにおける行数を測定して数値化したもの。空白行やコメント行のみ、括弧を閉じる“}”だけの行をカウントするかどうかがメトリクスによって異なる。開発物の規模という観点で見るのであれば、空白行やコメント行は不要であると考えられるため、それを除いた行数である「実行コード行数」を用いることが多い。こちらも用途によって使い分けることが必要になる。

・ファイル単位でのみ測れるメトリクス

例）定義されるクラス数、定義される関数の数

この二つのメトリクスは、その名の通りである。「クラス」や「関数」はファイル内で定義されるため、それらの定義数を測る場合にはファイル単位から測定することができる。これらはファイルよりも大きな単位でなければ測ることのできないメトリクスである。

これらのメトリクスについて、ヒストグラムを作成してどのような傾向があるかを観察した。

また、今回扱う実プロダクトにおいては、「不具合件数」と「納品回数」がデータとして得られる。「不具合件数」からは、プロダクトの規模に対する「不具合密度」も定義した。これらをそれぞれ以下のようなメトリクスである。

・不具合件数…「不具合管理台帳」に記述されている不具合の件数。台帳は、プロダクト大分類であるプロダクト A, プロダクト B の B 社開発分と C 社開発分, プロダクト C の４つに分けられている。そのうち、台帳の項目にある「分類」などを参考に細かいプロダクト分類のどれで起きた不具合であるかを分け、それぞれの件数としてカウントした。「分類」にはこれらの細かいプロダクト以外

に「インストーラ」や「B1 と B2」といったような不具合発生個所が複数にわたるものはカウントしていない。

・納品回数…一番初めに受け入れを認めた時を１回目とする。その後不具合が見つかって手戻りが発生した場合、一度手戻りが発生して修正が加えられ、納品が発生するたびに回数を増やす。最終的に納品完了となった時の総納品回数を測定した。

・不具合密度…上記の「不具合件数」を、各プロダクトの小分類のソースコード規模で割ったもの。ソースコード規模は、Understand で測定されるメトリクスの「CountLineCode」を用いた。これは、コメント行や空白行を除いたコメントの総行数を示す。
これらのメトリクスと、Understand の測定値として得られるソースコード規模や複雑度との相関関係を観察し、どのような知見が得られるかを調査した。

4.2.2. 不具合原因分類

次に不具合原因分類の方法について述べる。前述した不具合管理台帳には、不具合の詳細な内容とその原因について自由言語で記述されている。今回はその自由言語を人手で読み取って不具合がどの分類にあたるかを自分が評価した。その際に、具体的に不具合分類のうちどれにあたるのか判断に困るものもいくつか見られた。また、一つの項目にいくつかの不具合が記述されている欄もあった。それらをどれに分類し、いくつの不具合数としてカウントしたかを詳しく述べる。

・実装中の記述誤り
“仕様書には「○○」と定義すべきであるとされている文字列が「△△」となっていた”といった内容の不具合がいくつか見られた。これは「実装」工程における「実装誤り」ととることもできるが、細かく考慮するのであれば「データの誤り」であると判断したためこちらでカウントしている。

・原因に「考慮不足」と記述されているもの
“実際にこのプロダクトを利用する際には、このような機能が必要である”という実用的な面の考慮ができておらず修正すべきだと記述されている不具合もいくつか見られた。「考慮不足」という言葉から、本来設計段階で組み込んでおくべき機能が見落とされていたと判断し、「設計」工程の「機能設計の欠如」

に分類した。

・非バグと記載された項目
　一度は不具合であると判断され、不具合管理台帳に記述はされているものの最終的にはバグでないことが認められたものが存在した。そちらは「状態」という項目に「非バグ」との記述があったため台帳に記述はされているものの今回のカウントから外している。

・一つの項目にいくつかの不具合が記載されているもの
　これにあたる不具合は、ある一つの動作を実行したとき同時にいくつかの不具合が起きた時に記述されているものだった。ある箇所を一つ修正することで同時に問題が解決するような場合には、まとめて一つの不具合としてカウントし、いくつか別の原因が考えられるのであれば、その原因の数だけカウントすることとした。また、一度は修正されたものの別の不具合との兼ね合いで再発したというケースも見られた。その場合には、再発した回数だけ不具合件数をカウントしている。

4.2.3 元測定値と前後差分値の関連の調査方法

　まず元測定値と前後差分値の関係を調査するため、３次元グラフへのプロットを行う。X 軸に元測定値、Y 軸に前後差分値をプロットする。Z 軸には、{元測定値, 前後差分値}の組み合わせを満たす関数がいくつあるかをプロットする。３次元グラフは１プロタクトごとにそれぞれ作成し、各プロダクトの委託先変更の有無を考慮して傾向比較を行う。
　また、傾向分析の結果として差があるように見られたメトリクスについては２要因による分散分析を用いて定量的に評価する。２要因による分散分析を行うため、関数を２要因において分類する。一つ目の要因は「委託先変更の有無」である。委託先変更が行われたソフトウェアに含まれる関数ならば「True」とし、そうでなければ「False」とした。二つ目の要因は、「元測定値が品質基準を満たしているかどうか」である。この場合における品質基準は、メトリクスごとに異なるため対応する閾値を適宜決定する。品質基準となる閾値を決定した後に、閾値を満たしていれば「True」とし、満たしていなければ「False」に分類した。この２要因が前後差分値に影響を与えているかを調査することが目的であるため、前後差分値は数値データをそのまま扱った。

4.2.4 不具合修正回数の測定方法

本研究によって得られた傾向が、不具合の修正回数と関連しているかを調べるため、外部委託先との報告で用いていた文書（以下文書 A）を用いて修正回数の測定を行う。この文書に記述されているのは、図 1 にある「modifications」の期間に修正された履歴である。そのため、「development」工程における不具合修正回数は測定することができない。文書 A には、各リリースにおいてどのファイルを修正したかが記述されている。本研究では、各ファイルが何度のリリースにおいて修正されたかを測定することで「不具合修正回数」とした。

4.3 結果

以上のデータと分析の方法から得られた結果を述べる。

4.3.1 メトリクス分析結果

Understand によって測ったメトリクスを、ファイル単位で測定した。続いて、それらのヒストグラムを作成し調査を行った。まず一例として、開発規模の大きさがほぼ同程度である B2 と C4 における「CountLineCode（以下 CLC と表記する）」の分布図をそれぞれ図 4 と図 5 に示す。

これを見ると、多くは一番小さな区分に集まり、大きな区分になるにつれて頻度が低くなっている結果が得られた。また、どちらのプロダクトにおいてもひとつ突出して大きなファイルが少しあるという結果を示した。プロダクト規模の違う他のプロダクトにおいても同様の分布図を作成したが、おおむね同じような傾向を示したためそれらの結果は割愛する。

次に、同じようにプロダクト B2、C4 における「AvgCyclomatic（以下 AC と表記する）」の分布図を図 6 と図 7 に示す。

これを見ると、B2 における分布が少し右によってはいるが、基本的には「CLC」に似た傾向を示している。こちらのメトリクスについても、他プロダクトにおける調査を行ったが同じような傾向となっていた。

このように、行数に関するメトリクスや複雑度に関するメトリクスは、多少の差がありながら同じような傾向を示す結果となった。しかし、その中で一つ他のメトリクスらとは傾向の違うものが見つかった。そのメトリクスは、「RatioCommentToCode（以下 RCTC と表記する）」というメトリクスである。このメトリクスの分布図を図 8 と図 9 に示す。これを見ると、前述した二つのメトリクスとは全く傾向が違うことが見て取れる。「RCTC」は、「CLC」に対

するコメント行数の比率を表している。「CLC」の測定値はコメント行を含まないため、この比率が 1.0 を上回ることもあり得る。「RCTC」の分布は、両プロダクトにおいて 0.4 から 0.6 の区分が一番多く、その前後に均等に分布している。

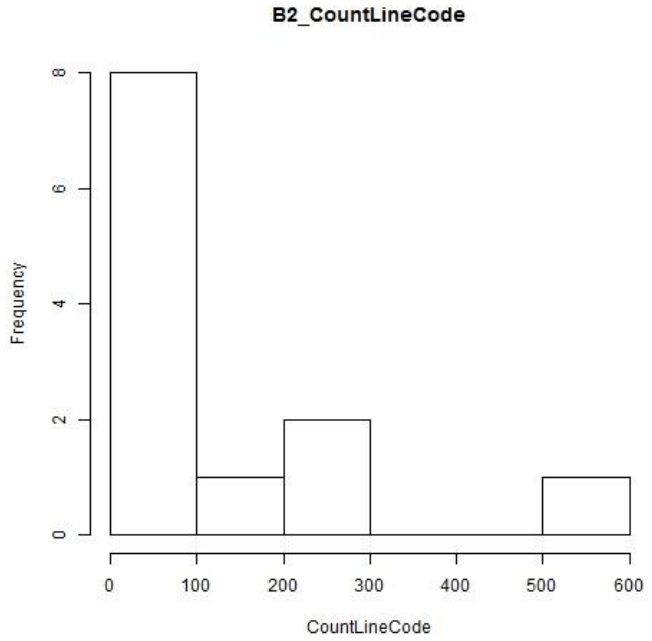

図 4　CountLineCode の分布図（B2,ファイル単位）

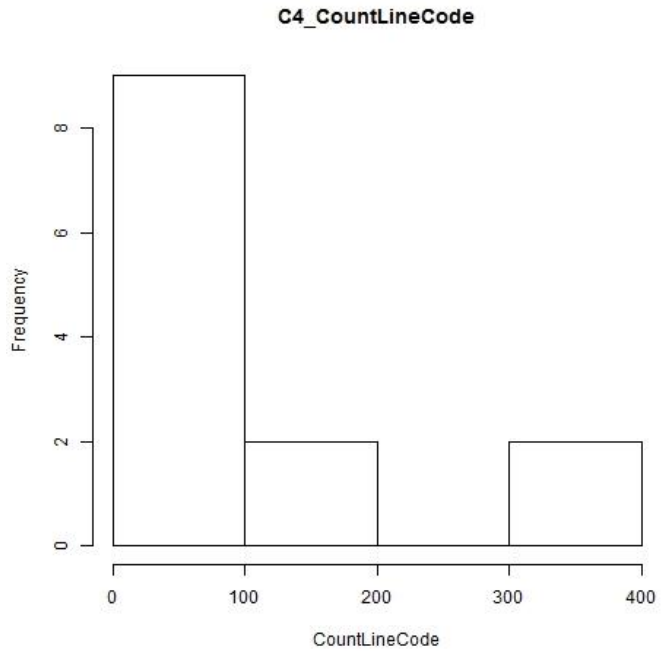

図 4　CountLineCode の分布図（C4、ファイル単位）

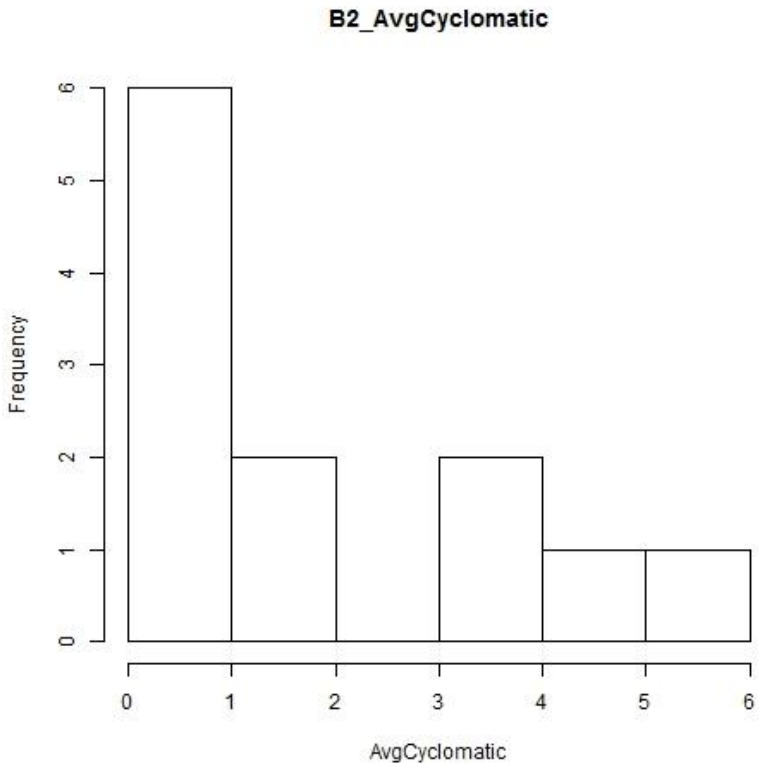

図 5　AvgCyclomatic の分布図（B2,ファイル単位）

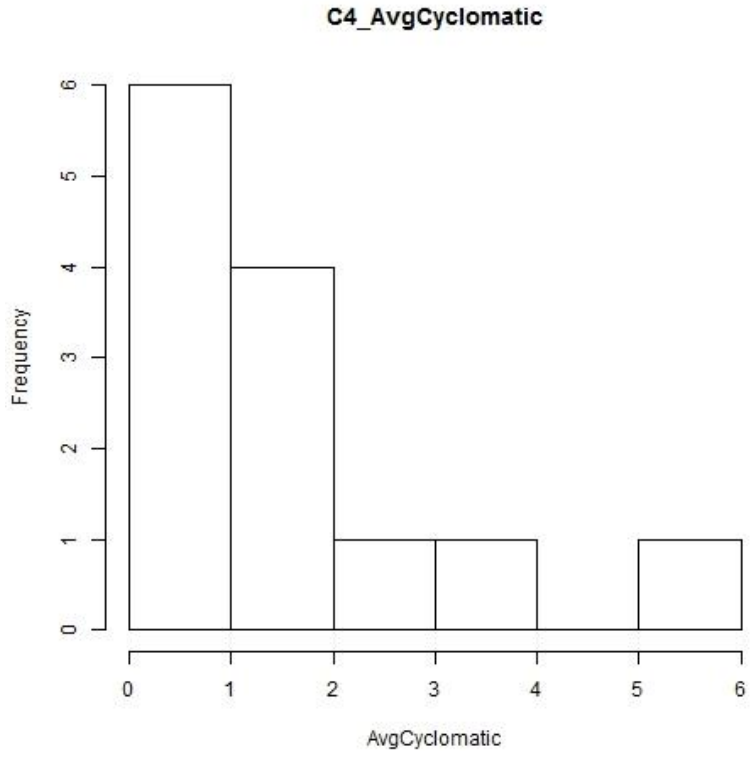

図 6　AvgCyclomatic の分布図（C4,ファイル単位）

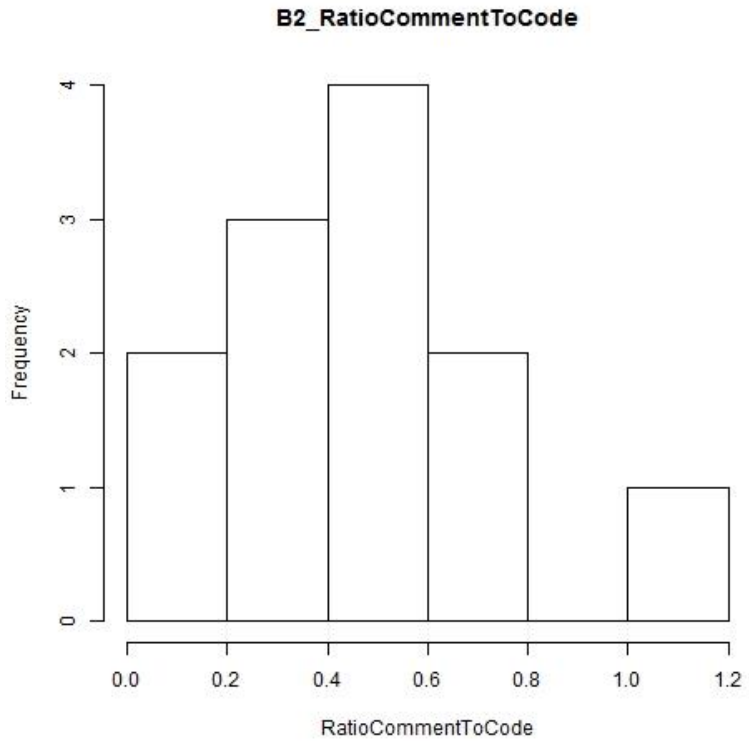

図 7 RatioCommentToCode の分布図（B2、ファイル単位）

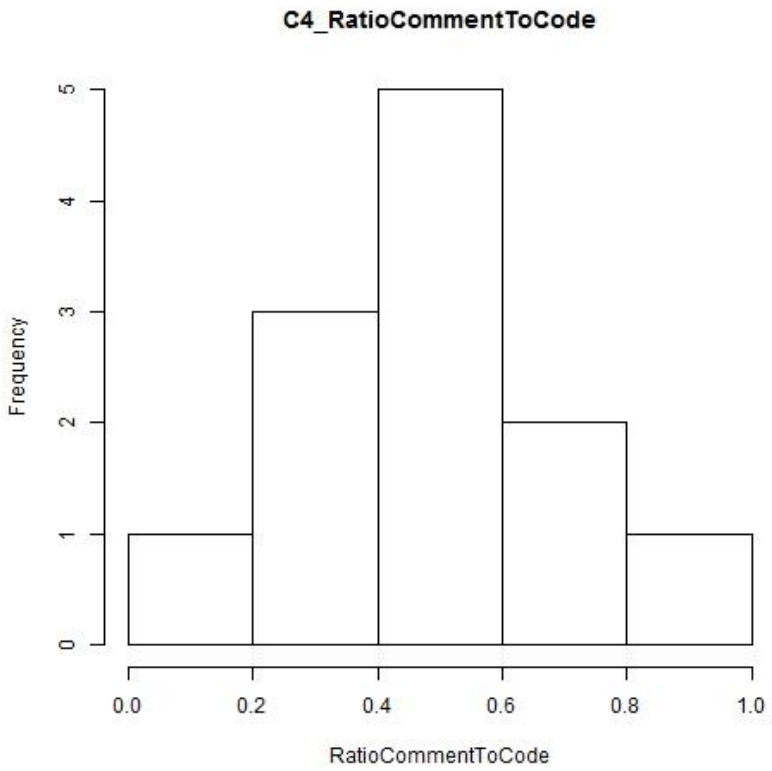

図 8 RatioCommentToCode の分布図（C4、ファイル単位）

次に、不具合密度や納品回数、プロダクトメトリクスどうしの関係を調査するために行った分析を述べる。この分析は散布図を作成して関係をみた。それらの結果の中でも特に知見が得られると思われる 2 つの散布図結果を示す。一つは、図 9 に示した「CLC」と「SumCyclomatic（以後 SC と表記する)」の散布図である。「SC」は、そのプロダクトに記述されている関数などの Cyclomatic 複雑度の総数である。２つ目は、図 10 の「CLC」と「BugDensity（以下 BD と表記する)」の散布図である。「BD」は、前述した「不具合密度」である。

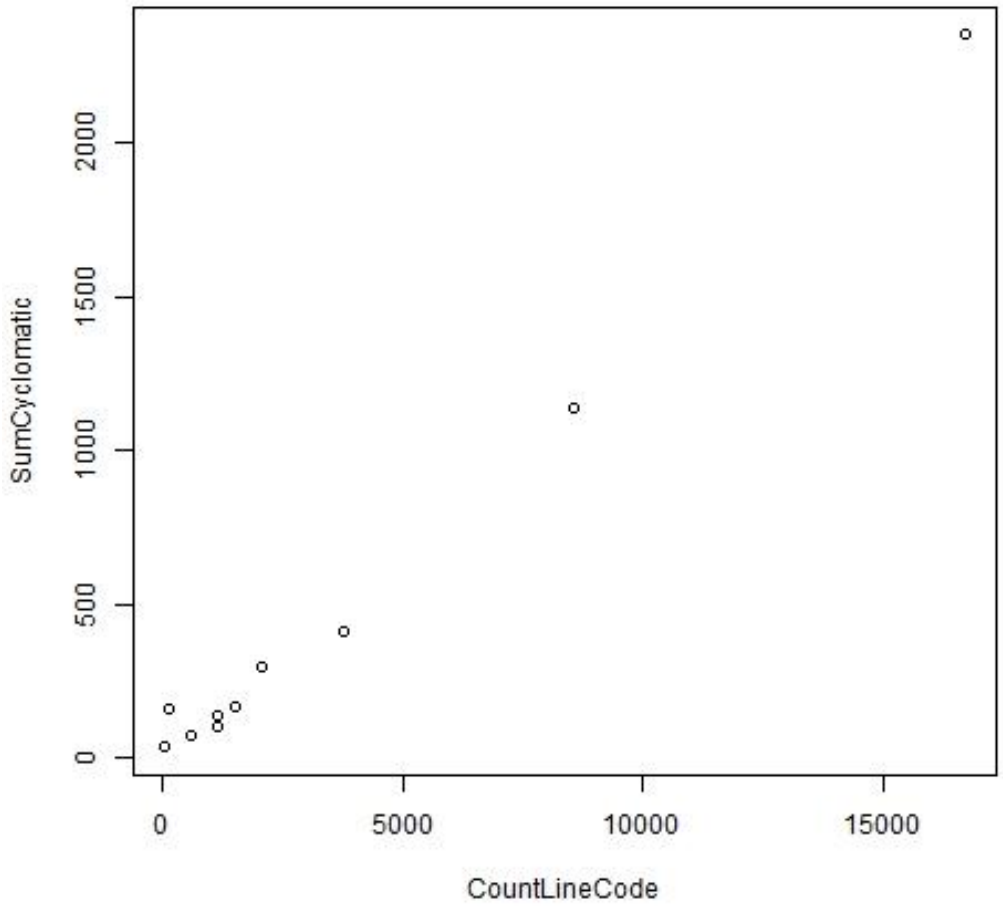

図 9 CLC と SC の関係を示した散布図

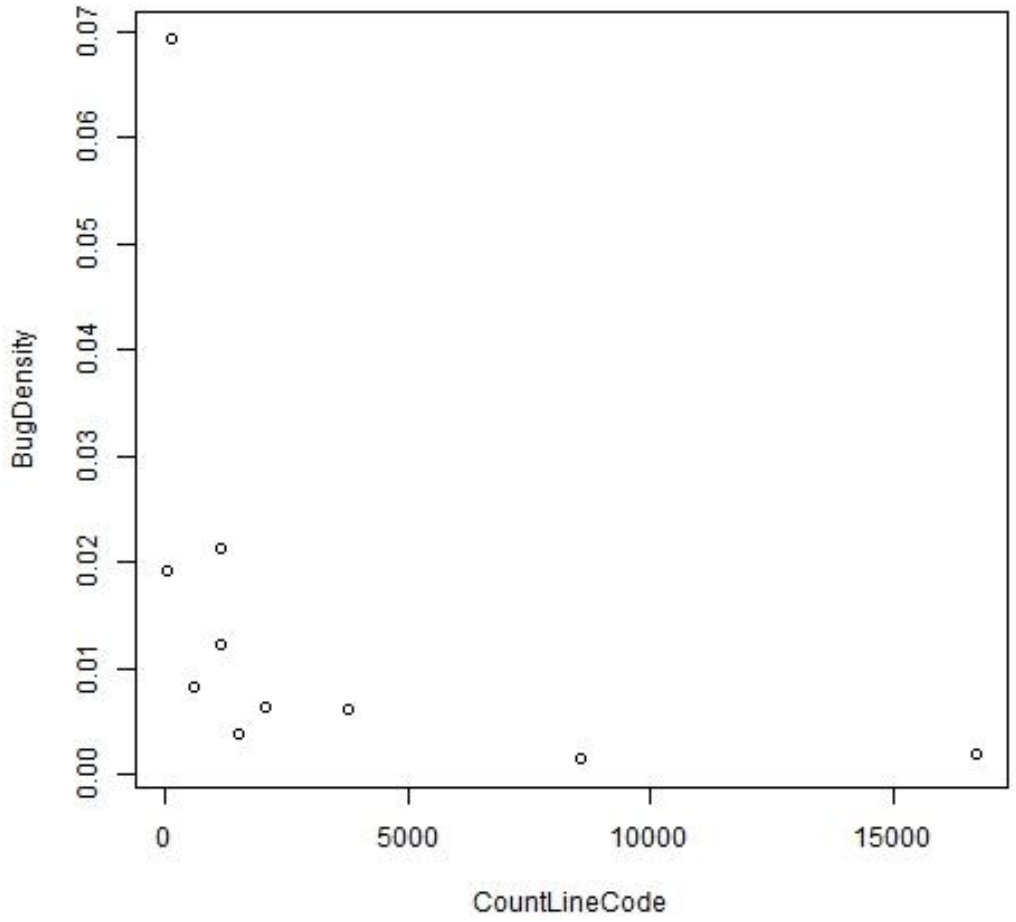

図 11 CLC と BD の関係を示した散布図

まず図 10 を見ると、非常に正の相関関係が高いことがわかる。相関係数は、0.9957076 となった。「CLC」はそのプロダクトの規模を示すものであるから、この結果は「規模が大きくなればなるほどコードの複雑度は大きくなる」ということを示しているといえる。

次に図 11 を見る。この散布図では、反比例のグラフのような曲線を描いているような結果となった。規模が非常に小さい場合には「BG」は非常に大きな値となり、規模の大きさが一定以上になるにつれて「BD」が収束していっている。

本論文では、RQ として以下の二つをたてた。

RQ1 外部委託開発をしたソフトウェアに見られるメトリクス測定値の共通の傾向はあるか

RQ1 についての回答は以下の点が挙げられる。

・ファイル単位のメトリクスのほとんどにおいて、大部分は一番小さな区分に集中し、数ファイルは突出して大きな値を示した。
・コメント率というメトリクスにおいては、どのプロダクトでも平均的に大きな値を示した。
・開発規模と複雑度の関連において、非常に大きな相関が得られた。
・不具合密度は、開発規模が大きくなればなるほど収束していく傾向にある。

4.3.2 不具合原因分類結果

次に、人手評価による不具合原因分類の結果について述べる。まずは、工程別に分類した結果を示す。この結果は、プロダクト別の結果をすべて表記するのは冗長になりすぎることと、プロダクト別に分ける必要のない結果であると考えられる。そのため、すべてのプロダクトの不具合をまとめたグラフで示す。以下の図 12 がその結果である。

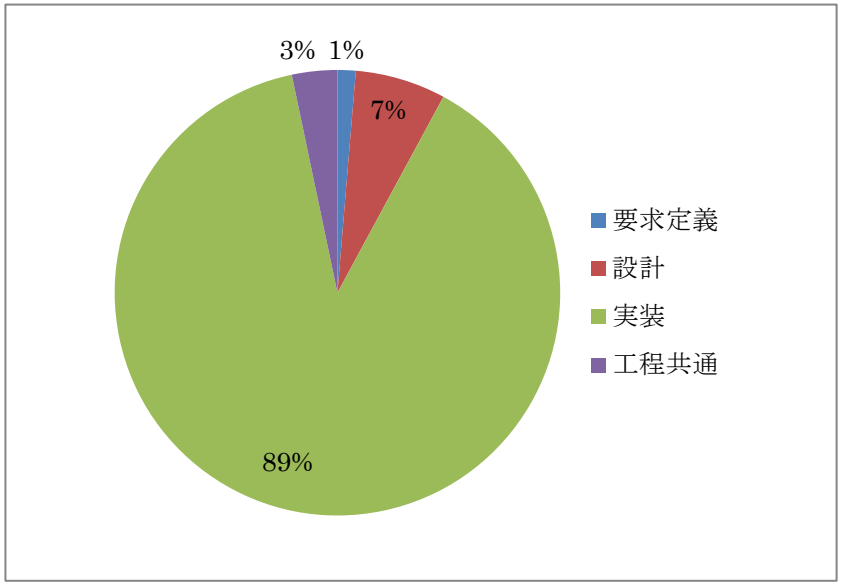

図 12 工程別の不具合原因分類

これらを見ると、ほとんどの不具合分類は「実装」工程で組み込まれたものであることがわかる。「要求定義」や「設計」を自社で行い「実装」を外部に委託しているプロダクトにおいて、この結果は「要求定義」や「設計」に大きな問題がなかったこと推測できる。

次に、特にその「実装」工程の中でも何が原因で不具合と分類されたのかを図 13 に示す。こちらの結果も、プロダクト別に比較などを今回は行わないためすべてをまとめたものを示す。

これらの結果を見ると、「機能の欠如」と「機能の実装誤り」が共に多く、次に「データの誤り」といった原因が多いことを確認できる。また、台帳に自由言語で記述されている「原因」の項目には「考慮不足」や「仕様誤認識」といった内容のものが特に多くみられた。

RQ2　一度納品された後に検出された不具合の原因について、複数のプロジェクトでどのような共通点があるか

以上の結果から、RQ2 についての回答は以下の点が挙げられる。

・一度受け入れを行った後に検出される不具合のほとんどは「実装」工程で埋め込まれたものである。

・外部委託開発という環境であるがゆえに、プロダクトに対する仕様の誤認識などが起こりやすい可能性がある。

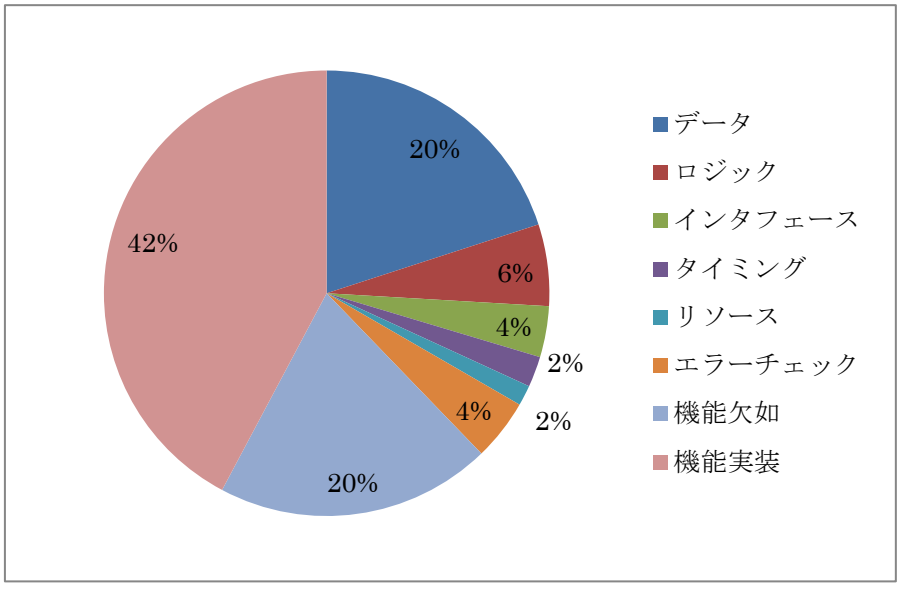

図 13 「実装」工程での不具合原因分類結果

4.3.3 開発委託先変更プロダクトの調査結果

本稿にて、事例における開発委託先変更プロタクトの統計解析結果を述べつつ、我々がたてた RQ の真偽について議論する。

RQ3　元測定値と前後差分値の関係は、開発委託先変更の有無によって違いがあるか？

提案の項では、「元測定値」が品質基準を満たしているならば、委託先変更による技術者の暗黙の知識やギャップは発生しづらいと仮説を立てた。この仮説が正しいならば、企業が委託先の変更を伴う場合でも、品質基準を満たすよう修正を施すことで、品質低下のリスクを低くすることができる。この仮説が正しいかを確かめるために、我々は測定した各メトリクスの元測定値と前後差分値の関係と委託先変更の有無による違いを調査した。

図２（3Dplot グラフ）は、測定したメトリクスの中でも特に顕著な傾向を示した”Cyclomatic complexity”における元測定値と前後差分値の関係である。本グラフでは X 軸が元測定値、Y 軸が前後差分値、Z 軸は元測定値と前後差分値の関係が同値の関数数を示している。productA と productB は、委託先変更を伴わなかったプロダクトであり、productC と productD は委託先変更を伴って開発されたプロダクトである。これら４つのグラフを「委託先変更の有無」によって比較すると、productA と productC において「前後差分値」が大きな値を示す箇所の傾向が異なっていることがわかる。それぞれのグラフにおける赤い丸で囲った部分を見ると、委託先変更を伴った productA と productB は元測定値の大きな関数において前後差分値が大きな値を示しているのに対し、委託先変更の無かった productC では元測定値の小さな関数において前後差分値が大きな値を示している。委託先変更の無かったもう一つのプロダクトである productD は、そもそも元測定値が大きな値を示すものがなく、前後差分値も大きな値を示す関数が存在していない。

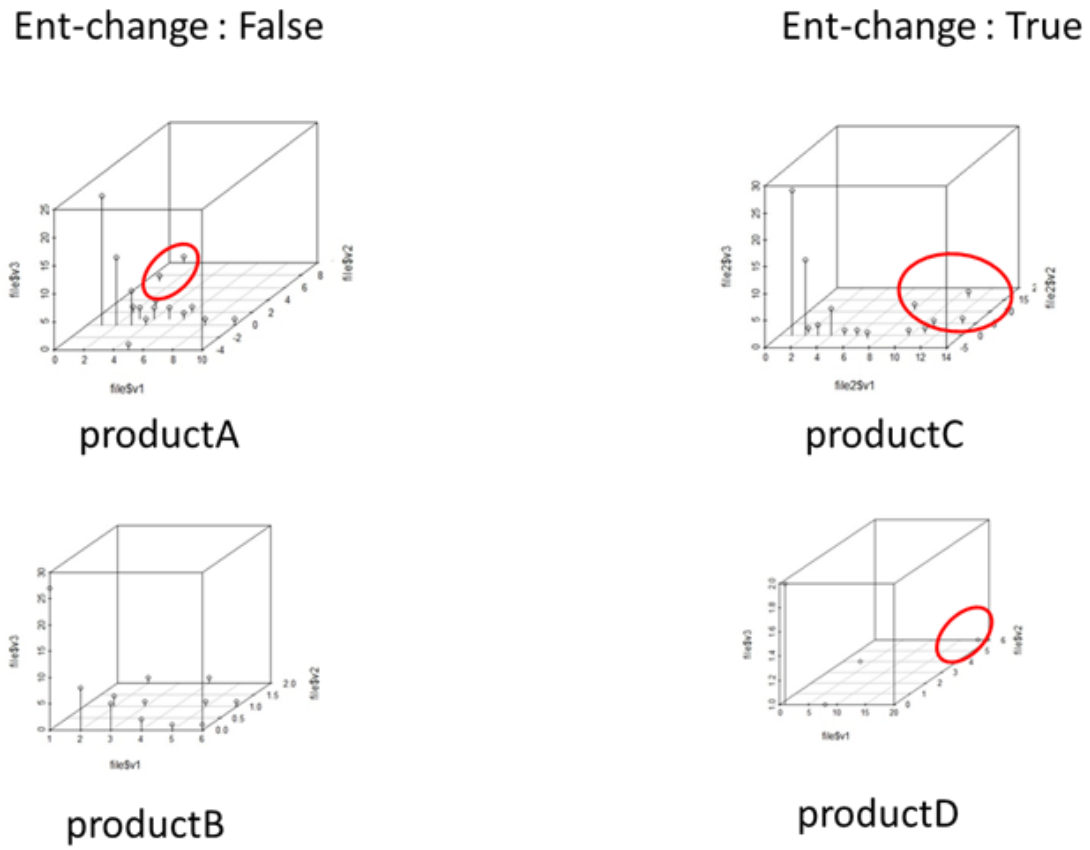

図 14 Cyclomatic completixy の 3 次元プロット

以上の結果から、元測定値と前後差分値の関係は「委託先変更の有無」によって異なるといえる。異なることを統計的に示すため、「委託先変更の有無」と「元測定値が品質基準を満たしているかどうか」の 2 要因を用いて「2 要因による分散分析」を行った。今回の分析は”Cyclomatic complexity”が「元測定値が品質基準を満たしているかどうか」の閾値を定める必要がある。この閾値は 10 とした。分散分析の結果を表 4 に示す。

表 4　2 要因による分散分析結果

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	
Ent-change	1	5.6	5.56	2.444	0.11939	
CCFactor10	1	120.8	120.8	53.124	5.62E-12	***
Ent-change : CCFactor10	1	24.7	24.73	10.877	1.14E-03	**
Residuals	219	498	2.27			

表 1 では、一番左の列にあるそれぞれの要素と前後差分値の関係をそれぞれの行で示している。注目するべきは Pr の列である。Pr はそれぞれの要素が前後差分値と関連がないといえる割合を示す。まず「委託先変更の有無」を示す Ent-changes の要素を見ると前後差分値の関連が無いといえる割合は約 11.9%

であることがわかる。この値は統計的に関連があるといえる数値ではない。次に「元測定値が品質基準を満たしているかどうか」を 10 という閾値で分類した要素であることを示す CCFactor10 について見ると、5.62×10^-10%であり関連がないといえる割合は非常に低いことが確認できた。最後に、この２つの要素を両方考慮したときの関連性は 0.114%でないといえる。これらの結果から、我々が提案した「元測定値」と「前後差分値」の関連があること、この関連が「委託先変更の有無」によって異なる結果を導くという結果を統計的に示すことができた。

RQ4　元測定値、前後差分値と不具合修正回数の間にはどのような関係があるか？

RQ5　元測定値、前後差分値と不具合修正回数の間にある関係は、開発委託先変更の有無によって違いがあるか？

次に、RQ3 の調査結果と修正回数にどのような関係があるかを評価するため RQ4 と RQ5 について調査する。RQ1 への回答として、委託先変更を伴う場合は元々複雑な関数がより複雑になりやすい傾向が存在し、委託先変更を伴わない場合はこのような傾向はなく品質低下のリスクが低いという結果が得られた。本セクションでは、このような傾向を示した関数は何度の修正を経て前後差分値が増加する結果となったのかを調査することで、修正を引き起こす原因と修正が引き起こす前後差分値の変化を明確にする。調査結果を以下の表に示す。

表 5 前後差分値の大きいメソッドの不具合修正回数

product	File	Method	元測定値	前後差分値	Ent-change	modification	Ranking
A	1	a	1	10	FALSE	4	2nd
	2	b	1	7	FALSE	8	1st
C	1	a	11	16	TRUE	11	1st
		b	13	5	TRUE		
	2	c	8	11	TRUE	6	3rd
D	1	a	18	6	TRUE	5	1st

この表における Product は、各 File と method がどのプロダクトに属しているかを示す。表に示した method はすべて”Cyclomatic complexity”の前後変化量が５以上のものであり、前後変化量と元測定値の列はそれぞれの method における”Cyclomatic complexity”の値を示している。「Ent-change」の列は委託先変更の有無を示しており TRUE ならば委託先変更有のプロダクト、FALSE ならば委託先変更無のプロダクトに属していることを示す。「modifications」の列は各ファイルの不具合修正回数を示す。これは測定の都合上 method 単位に分けることができないため、ファイル単位で示している。「Ranking」の列は、各プロダクトにおいて修正回数が多い順にファイルを並べた時、上位何番目の

ファイルであるかを示している。これらの結果を見ると、前後差分値が大きな関数は軒並み修正回数の多いファイルに含まれていることがわかる。

以上のことを考慮して考察すると、まず「元測定値の大きさは修正回数と関係がない」といえる。なぜならば、「前後差分値の大きな関数における元測定値」は RQ1 の結果から「委託先変更を伴う場合は元測定値が大きく、伴わない場合は元測定値が小さい」のにもかかわらず、前後差分値が大きな関数は軒並み修正回数の多いファイルに含まれているためである。次にいえるのは、「修正回数の多いファイルは、複雑な関数を含んでいる可能性が高い」ということである。なぜならば、前後差分値が大きな関数は軒並み修正回数の多いファイルに含まれていることが調査結果から確認できるためである。修正回数を数えるタイミングは「Second development」の初版リリース時点からであるため、「Second development」の初版リリース時点でのメトリクス値を測定することでより詳細な分析ができるかもしれない。

これらのことから、RQ2 と RQ3 に対しては以下のように結論付けることができる・

RQ4　元測定値と修正回数には関係がなく、修正回数の多いファイルには前後差分値の大きい関数が含まれている可能性が高い。

RQ5　委託先変更の有無に関わらず、前後差分値の大きな関数を含むファイルにおいてコードの修正が何度も行われていた。

4.4 考察

以上の結果から、外部委託開発をしたプロダクトにおける知見として何が得られるかを考察する。

4.4.1 メトリクスから得られる知見

まず、プロダクトメトリクスの分布図の多くが「一番小さい区分に集中し、いくつかのファイルが突出して大きな値である」という傾向が得られた。この傾向はファイルごとの責務が一部の大きなファイルを除いて平均的であると考えることができる。いくつかの値が突出したファイルの責務を、他ファイルに少しずつ分割することでより責務が大きすぎるファイルを少なく抑えることができそうである。別の外部委託開発先であるのにも関わらず同様の傾向になったことには何らかの意味がある可能性がある。

次に、他メトリクスの傾向とは違う結果を示した「RCTC」について考える。外部委託をした開発物における「コメント行」というのは、依頼元の企業で内部のデータを確認した際に、構造やコーディングの意図が容易に確認できるよ

うにするためではないかと考えた。コメントによって説明が加えられていれば、依頼元の企業が手を加える際の理解しやすさが変わるだろう。つまり、「RCTC」が大きいほど見たときにわかりやすいコードであると仮定できるかもしれない。しかし、一言に「コメント行」といってもその種類は様々である。開発環境によってはコメント行が自動で生成される場合もあり、それが数値に大きく影響している可能性がある。このメトリクスについて深く調査するのであれば、コメントの分類分けをして、有用な情報として記述されているコメントだけを抽出するといった方法を確立すればより精度の良い結果が得られる可能性がある。

散布図を用いた調査では、二つの結果を示した。一つ目の「CLC」と「SC」の散布図では、非常に大きな正の相関があることが確認できた。これは「開発規模が大きくなればなるほどコードの複雑度は比例して大きくなる」ということを示している。直感的に予想できる仮定が、実際に結果としてあらわれており、これは一つの知見であるといえるだろう。これが外部委託開発の特徴であるかどうかを判断するには、オープンソースソフトウェアや外部に委託しない内製プロダクトにおいての傾向を見る必要がある。

二つ目の結果は、「CLC」と「BD」における関係である。開発規模が大きくなればなるほど不具合密度はある数値に収束し、開発規模が小さいものは不具合一件の重みが大きいため場合によってはとても大きな不具合密度を示してしまうことがあることを確認できる。このとき、開発規模がいくつを超えた時に不具合密度が収束し始めるのか、同じ開発規模でも不具合密度が大きく違うプロダクトではどのような違いがあるのかを、今後調査するべき指針として挙げられる。

4.4.2 不具合原因分類から得られる知見

不具合が最も多く埋め込まれている工程は「実装」工程であることを確認することができた。これは、依頼企業による「要求定義」や「設計」に抜けや漏れが少ないということを意味していると考えられる。

その「実装」工程の中でも特に多くの不具合原因として分類されたものは「機能の欠如」と「機能実装の誤り」である。また、台帳に記述された内容には「仕様誤認識」や「考慮不足」といったまえがきが書かれた不具合もしばしばみられた。なぜこのような結果になったかを考えると、「外部委託先がその開発物を実用するわけではない」からではないかと考える。開発を行うのは外部企業であり、依頼元はその時々で変化する。分野もさまざまであると考えると、外部企業が開発する際には、開発物に関する分野に精通していない技術者が開発することが多いと思われる。この場合、開発したものを実際の現場で利用する状

況を完全に考慮することは困難だろう。それゆえに、「仕様誤認識」や「考慮不足」といった原因から「機能の欠如」や「機能実装の誤り」が生じてしまうのではないかと考えられる。

一方で考えるべきは、今回データとして用いた「不具合管理台帳」のみで判断できない情報である。本研究では、依頼元の企業が作成したプロダクトの仕様書をデータとして用いていない。もしこの仕様書の記述がわかりにくいものであった場合、不具合の原因が外部委託先ではないという可能性も考えられる。こちらとの関連性も考慮したうえで、慎重に考察を行う必要がある。

4.4.3. 開発委託先変更の調査

本研究では、元測定値と前後差分値という２つの値に着目し、産業プロダクトを用いて統計分析を実施することで委託先変更の有無による２つの値の関連性を調査した。その結果、委託先変更を伴う開発では”Cyclomatic complexity”の元測定値が大きな値である関数において前後差分値が大きな値を示した。一方、委託先変更を伴わない開発では”Cyclomatic complexity”の前後差分値が大きな値を示したのは元測定値が小さい関数であった。これに加えてコードの修正回数という観点からも関連を調査した結果、委託先変更の有無に関わらず前後差分値が大きな値を示した関数は多くのコード修正がなされていた。

これらの結果を踏まえると、現場で開発を行う技術者は「委託先変更を伴う開発において、委託先変更を行う直前のプロダクト内で複雑度の大きな関数が委託先変更後により大きな複雑度になる恐れがある」という考えのもとで開発を行うことができる。そのため、委託先変更を行う直前のプロダクトにおけるメトリクス値を測定して、複雑度の大きな関数があったならばリファクタリングなどを行うことでより複雑になるリスクを防ぐことができる。

4.5 妥当性の脅威

まず挙げられるのは、独自の判断を加えた不具合原因分類の信頼性である。本論文では、参考文献にあったものから不必要なものを排除している。また、参考文献元の資料で説明しきれていなかった不具合原因について独自の判断で原因分類をしており、それが正しいという評価をできていないことが問題として挙げられる。

次に挙げられるのは、不具合原因の分類結果が「機能実装の誤り」と「機能の欠如」に集中したことについてである。人手の評価は自分が行ったことは前

述したが、この作業をしている間に気付いたことがいくつかある。それは、この不具合原因分類の粒度についてである。詳しく言えば「機能実装の誤り」と「機能の欠如」という項目は非常に粒度が大きく、広い意味でいえばその他の原因である「ロジックの誤り」や「タイミングの誤り」なども当てはまってしまうのではないかという懸念がある。この「機能実装の誤り」と「機能の欠如」という二つ以外の原因は具体的なものが多い。これらの細かい原因分類のどれにも分類されなかった場合には、「機能実装の誤り」か「機能の欠如」という原因に分類せざるを得ないことが多かった。

この問題の原因となっているのは、人手評価を行ったもの（今回の例でいえば自分）の知識不足である可能性がある。また、原因となっているのは不具合原因分類表そのものである可能性も挙げられる。まず人手評価の信頼性に関する問題を解決するためには、そのプロダクトに深く精通している方に評価を依頼するといった方法が挙げられる。不具合原因分類表の粒度が問題であるならば、参考文献のものをベースに新たな原因分類表を作成することが必要かもしれない。新たな分類表を作成する際には、今回得られた知見から分類すべきである不具合原因を議論する必要があるだろう。これを実行することによって、より詳細な原因傾向を見つけることができる可能性がある。

5. 関連研究

5.1. 外部委託開発物の評価という観点

筆者の知る限りでは、外部委託開発の中でもとくに「ニアショアアウトソーシング」した際の実プロジェクトを対象に調査を行った研究は見つからなかった。一方で、「ニアショアアウトソーシング」と関係としては対になる「オフショアアウトソーシング」に関する研究はいくつか見られたので関連しそうなものをいくつか紹介する。

文献[]では、オフショアリング先を決めるにあたって、独自に「オフショアリング成功度モデル」を開発した。これを用いることでオフショアリング先を評価し、どの地域や国へオフショアリングを行うかを分析している。対象となる国はフィリピン、インド、エジプトであり、結論としてはこれら３か国にオフショアリングすることは有効であるとなっている。この「オフショアリング成功度モデル」は、専門家へのインタビューや今まで積み重ねてきた調査結果を用いて、コミュニケーション頻度や言語能力といった国の要因。仕様の明確さや納期の切迫度といったプロジェクトの要因を数値化して評価式に組み込んでいる。

本研究の対象はニアショアであるため、国の要因といった属性は存在しないが、過去にニアショアを行った企業について、いくつか評価項目を用意してプロジェクトに対する評価を人手で行って次にオフショアを行う際の指標として用いるというようなことは可能かもしれない。

5.2. プロダクトメトリクス分析という観点

次に外部委託開発という観点ではなく、プロダクトメトリクスの静的解析を用いて新たな知見を得るという共通点をもつ研究について紹介する。

今回得られた特徴に、他メトリクス値と比較したときのコメント率の高さが挙げられる。このコメント率に関連する研究として、オープンソースソフトウェアを対象にしたコメント記述、コメントアウト率と潜在する不具合率がどのような関係にあるかを明らかにした研究がある。[]　考察の項でコメント率の比較対象として出した結果はこの研究で用いているものである。この研究では、コメント記述率をあらわす「CMR」と、コメントアウト率を示す「COUTR」という二つのメトリクスを用いている。この二つのメトリクスの両方を使って潜在不具合率を比較した場合と、「CMR」のみと「COUTR」のみを用いた場合の三パターンで潜在不具合数とこれらの関係を調査している。結果としては、

“コメントが 12.6%を超える割合で記述されているようなコード（コメント記述率＞0.126）では、コメントが全くされていないコードに比べてフォールト潜在の可能性が 2 倍程度（正確には 1.78~2.02 倍）高い”というものであった。

本研究で用いたデータでは、コメント率の分布が文献[]のものよりも大きな値に多く分布していた。そのため、文献[]と同様の手法を用いた場合にはまた違う結果になることが期待できる。この結果を比較することによって、オープンソースソフトウェアと外部委託開発物による違いを定性的に評価できるかもしれない。

6. おわりに

6.1. 今後の展望

今後行うべき調査としては、プロダクトメトリクスをプロダクトやファイル単位のものからさらに集約して、クラス単位や関数単位での測定値を用いるものである。これをすることによって、プロダクトやファイル単位のメトリクスでは丸められてしまっていた可能性のある傾向を詳細に見ることができる。

また、このような外部委託開発をしたプロダクトとの比較のために外部委託を行わず自社で開発した内製のプロダクトや OSS といったプロダクトの分析を行うことで、より外部委託開発プロダクトにおける特徴を明確にすることができる。

6.2. まとめ

年々大規模化しているソフトウェアの開発コストを削減するためには、外部委託開発という手段が用いられる。しかし、この手段を用いたときには開発されたプロダクトの品質保証がされていない。そのため、外部委託開発物に特化した、評価指標を作成することを考える。この評価指標を作成するにあたって、実際に過去に外部委託開発されたプロジェクトのデータを用いて不具合の原因やメトリクスの特徴を調査した。それによって、評価指標を決める参考となるいくつかの知見や、今後の課題を整理した。

謝辞

本研究を進めるにあたり、数々のご指導を頂いた早稲田大学の深澤良彰教授・鷲崎弘宜准教授に深く感謝いたします。

共に研究に励み、様々な面でご協力頂いた深澤研究室・鷲崎研究室の皆様に深く感謝致します。

参考文献

[1]名倉正剛, 田村清朗, 川口真司, 田中眞吾, 飯田元, 事例紹介：OEMソフトウェア製品検証工程, SEC journal Vol.8 No.4, Dec 2012
[2] 関口和代, アウトソーシング・ビジネスの現状と課題, 東京経大学会誌第270号, pp143-157, 2010
[3]倉下亮, 吉村博昭, 野中誠, 誉田直美, CKメトリクスの分布に基づくソフトウェア設計の質の定量的分析, SQiPシンポジウム, 2011
[4]阿萬裕久, 野中誠, 水野修, ソフトウェアメトリクスとデータ分析の基礎, コンピュータソフトウェア, Vol.16, No.5(1999), pp.1-13. 2010
[5] Understand, https://www.techmatrix.co.jp/quality/understand/
[6]松本隆明, SEC BOOKS 組み込みソフトウェア開発における品質向上の勧め[バグ管理手法編], 独立行政法人情報処理推進機関（IPA）, pp.46-50, 2013
[7]阿萬裕久, オープンソースソフトウェアにおけるコメント記述およびコメントアウトとフォールト潜在との関係に関する定量分析, 情報処理学会論文誌, Vol.53 No.2, 2011
[8] 稲垣康一, 中野冠, グローバル企業のソフトウェア開発におけるオフショアリング先の評価, プロジェクトマネジメント学会 2011 年度春季研究発表大会予稿集, 2011