Blocking Bugの発生原因を理解するための依存関係分析

金城 清史
和歌山大学 システム工学部
s161017@sys.wakayama-u.ac.jp

山谷 陽亮
和歌山大学 システム工学研究科
s151049@sys.wakayama-u.ac.jp

松本 明
和歌山大学 システム工学研究科
s151045@sys.wakayama-u.ac.jp

大平 雅雄
和歌山大学 システム工学部
masao@sys.wakayama-u.ac.jp

要旨

影響の大きい不具合の 1 つとして *Blocking Bug* がある．*Blocking Bug* とは他の不具合の修正を妨害する不具合のことである．先行研究では *Blocking Bug* の特徴を明らかにしている．しかし，発生の原因を分析したものではない．本論文では *Blocking Bug* の早期発見，発生の予防を目的として，*Blocking Bug* の実態把握と発生原因を理解するための分析をおこなった．分析の結果，*Blocking Bug* は通常の不具合に比べて修正のために変更するファイルが多いことや，多くのファイルに参照されていると *Blocking Bug* の発生の原因になることを明らかにした．

1. はじめに

ソフトウェア開発では，ソフトウェア保守に要するコストがソフトウェアライフサイクル全体に要するコストの大半を占めることが知られている [1, 2]．そのため，ソフトウェア保守の改善支援を目的とする研究が盛んにおこなわれてきた．特に，Mining Software Repositories (MSR) 分野では，主に不具合管理システムに蓄積された不具合データの解析結果に基づいて，不具合修正および不具合管理プロセスを支援する手法が提案されている．例えば，不具合修正タスクの最適割り当てを支援する手法 [3] や不具合修正タスクの優先順位や重要度を推定する手法 [4, 5] ，重複して報告された不具合を検出する手法 [6, 7] などが挙げられる．

先行研究の多くは不具合修正作業および管理プロセスの効率化を目的とするものの，個々の不具合の特徴や影響範囲を十分に考慮したものではない．しかし，1 つの不具合がシステム全体の挙動を不安定にさせたり，場合によっては致命的なエラーにつながることもあるため，重大な問題を招く可能性のある不具合を考慮することが重要である．

これまでの反省もあり，特に最近では，個々の不具合がユーザの満足度や開発スケジュールの変更などに与える影響の大きさを考慮する研究が増えている．保守プロセスに充てられるコストやリソースには限りがあり，全ての不具合を同等に扱うのではなく，High Impact Bug[8] と呼ばれる影響度の大きな不具合 [9, 10, 11] を優先的に修正することの必要性が認識され始めたためである．

High Impact Bug の内，本研究では特に，Blocking Bug[12] を対象にする．Blocking Bug とは，他の不具合修正を妨害する不具合である．Blocking Bug が修正されない限り，関連する他の不具合 (Blocked Bug) も修正することができないため，Blocking Bug は優先的に修正されるべき不具合であるとされている．

Garcia ら [12] の研究では，新たに報告された不具合が Blocking Bug かどうかを予測するモデルを提案しているが，Blocking Bug が生じる原因や予防する方法については述べられていない．そのため，本研究では，不具合報告の依存関係，モジュールの依存関係，開発者の依存関係といった Blocking Bug の発生原因となり得る依存関係を分析し，Blocking Bug の早期修正と予防を支援するための知見を得ることを目的とする．

2. 修正活動における Blocking Bug の問題点

2.1. Blocking Bug 定義

Blocking Bug とは，他の不具合修正を妨害する不具合である．

不具合管理システムに報告された不具合が Blocking Bug であることが判明した場合，Blocking Bug と不具合修正を妨害されている側の不具合 (Blocked Bug) との間に関連付けがおこなわれる．この際，不具合管理システムでは，二つの不具合間の関連付けのために，Blocking Bug の内容が記述された不具合報告には “blocks” というタグが付与される．同様に，Blocked Bug の内容が記述された不具合報告には “is blocked by” というタグが付与される．なお，タグ付けは自動的におこなわれるのではなく，不具合管理システムの管理者らによっておこなわれる．

多くの不具合管理システムには，不具合修正に際して依存関係のある不具合同士を関連付けて管理するための機能が備わっている．本研究では，依存関係のある Blocking Bug と Blocked Bug のことを Blocking Set と呼ぶこととする．また，ソフトウェアの不具合は，複数のファイルが関連しあって 1 つの不具合の症状として現れる場合や，1 つのファイルが原因となって複数の不具合の症状を引き起こす場合があり得るが，本研究では，1 件の不具合報告に記録された症状を 1 つの不具合として取り扱う．したがって，本論文中で特に断りなく Blocking Bug という用語を使う場合は，不具合報告に記載された Blocking Bug の症状，あるいは，不具合報告そのものを指すこととする．

2.2. 問題点

Blocking Bug を予測するための Garcia ら [12] の研究では，いくつかのオープンソースプロジェクトを対象とした予備調査から，Blocking Bug は Blocking Bug ではない不具合（非 Blocking Bug）よりも多くの修正時間を必要とすることが示されている．リリース直前やプロジェクトの繁忙期であったり，緊急な対応を要する不具合修正をおこなっているときなど，十分な時間が確保できない場合に Blocking Bug が発見されると，ソフトウェアの品質を改善（あるいは維持）することができない可能性がある．これら Blocking Bug の問題を，図 1 を用いて整理する．

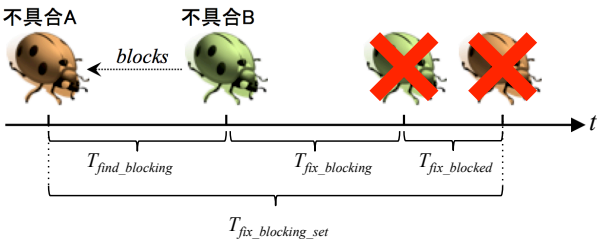

図 1. 不具合修正を阻害する Blocking Bug

報告された不具合 A は，不具合 B によって修正を妨害されており，不具合 B が先に修正されない限り不具合 A を修正することができない状態にある．つまり，不具合 A が甚大な影響を与える不具合であっても，不具合 B が修正されない限り不具合 A を修正することができない．したがって，不具合 A を修正するには，本来必要な時間 ($T_{fix\_blocked}$) よりも多くの時間 ($T_{fix\_blocking\_set}$) が必要となる．不具合修正の遅延は顧客満足度の低下に繋がるため，できるだけ早く Blocking Bug を検出する必要がある．

しかしながら，不具合 A の担当者は不具合 B を修正するためのコードやモジュールについて熟知しているとは限らず，不具合 B，すなわち，Blocking Bug の検出自体に時間 ($T_{find\_blocking}$) がかかってしまう場合もある．また，Blocking Bug である不具合 B を検出できたとしても，不具合 A と不具合 B の担当者が異なる場合は互いに情報交換しながら修正作業をおこなう必要があるため，通常の不具合よりも多くの時間 ($T_{fix\_blocking}$) を必要とする．実際，6 つのオープンソースプロジェクトを対象とした Garcia らの研究 [12] では，Blocking Bug は，非 Blocking Bug (Blocking Bug 以外の不具合）に比べ 15–40 日程度の修正時間が長くなるとされている．なお，Blocking Bug が検出されるまでの時間 ($T_{find\_blocking}$) は短い場合もあれば長い場合もあり (3–18 日)，プロジェクト毎に傾向は異なる．プロダクトの複雑度やコンポーネント間の結合度に関連があるように思われる．Garcia らの分析結果を受けて，Blocking Bug の発生原因を理解することが Blocking Bug を早期に検出したり予防したりするのにより重要であると考えるに至った．

3. 先行研究

3.1. Blocking Bug 予測

Garcia らの研究では Blocking Bug に関する特徴を分析し，新たに報告された不具合が Blocking Bug か否かを予測するモデルを構築している [12]．6 つの OSS プロジェクト（Chromium, Eclipse, FreeDesktop, Mozilla, NetBeans, OpenOffice）を対象におこなった分析では，Blocking Bug は非 Blocking Bug と比較して 2 倍から 3 倍ほど修正時間が長くなる（15 日から 40 日程度修正時間が長くなる）ことや，プロジェクトによって差はあるものの，検出にかかる時間は 3 日から 18 日程度であると述べている．

また，Blocking Bug かどうかを予測するモデルでもっとも高く寄与したメトリクスは不具合票に投稿されたコメントであることを明らかにしている（Mozilla のみ，その不具合票に関わりを持つことを望んだユーザ (CCList) の数がもっとも寄与している）．

先行研究の分析は，Blocking Bug の検出を目的としており，Blocking Bug の発生原因や発生の予防法については述べられていない．

本研究の目的は Blocking Bug の早期修正，発生の予防をおこなうことである．そのために Blocking Bug が発生する原因を理解する必要があり，発生原因を理解するための分析をおこなう点で先行研究と異なる．

3.2. 依存関係分析

ソフトウェア保守における依存関係の分析は盛んにおこなわれている．Biazzini ら [13] は， 87 プロジェクトを対象として，分散管理システムにおけるコミット履歴を基に共同開発における特徴を分析している．分析の結果，開発者の人数が多くなると複雑なブランチを形成することやブランチごとのコミット回数と開発者の共同作業には関係がないことを示している．

Bhattacharya ら [14] は，11 プロジェクトを対象にソースコードを基に作成したグラフと開発者を基に作成したグラフを用いて，ソフトウェアの開発支援のための分析をおこなっている．分析の結果，ソースコードの参照関係を基に作成したトポロジグラフのうち，入次数が多い，つまり多くのファイルから参照されているほど，そのファイルから発生するバグの重要度が高いことや，不具合の担当者とその不具合を修正できずに新たな担当者として不具合を担当した開発者間にエッジを結び作成したトポロジグラフにおいて，トポロジの変化が大きいほど不具合が発生しやすいことが確認された．

いずれもモジュール間でのみの分析や人のみの分析であり，本研究のように，不具合，モジュール，人を結びつけて 1 セットで分析した研究はまだされていない．

4. Blocking Bug の発生原因理解のための依存関係分析

4.1. 概要

プロジェクトに報告される不具合のうち Blocking Bug に特化した研究は，Garcia らの研究 [12] 以外には存在しない．また，Garcia らの研究は，Blocking Bug の予測を目的としたものであり，Blocking Bug の実態や発生原因を明らかにはしていない．

そこで本研究ではまず，Blocking Bug の実態を調査し，その後，Blocking Bug の発生原因理解のための分析を行う．Blocking Bug の発生原因理解のための分析にあたり，本研究では 3 つの分析視点を導入する．

不具合管理システムに報告された不具合は，まず適任の修正担当者（多くの場合，修正対象物の作者）に割り当てられる．次に修正担当者は，その不具合を発生させている原因を調査し，修正対象物（パッケージ，コンポーネント，ファイルなど）を決定する．この不具合解決プロセスは，Blocking Bug を対象にした分析においては非常に複雑なものとなる．複雑な関係は，修正すべき不具合が単独で発生したものであれば（Blocking Bug でなければ）発生しない．この点からも Blocking Bug の解決は通常の不具合修正に比べて容易なものでないことが分かる．したがって，分析の切り口としてまず，以下の 3 つの視点で複雑な依存関係を分解する．

- 不具合報告間の依存関係（Blocking Bug と Blocked Bug の依存関係）
- 修正担当者間の依存関係（修正作業を進める上での依存関係）
- 修正対象物間の依存関係（call-caller 関係などの依存関係）

これらの依存関係をそれぞれ分析することで，Blocking Bug の発生原因をある程度絞り込むことができ，Blocking Bug の予防する上での知見が得られるものと期待している．さらに，補足的な分析視点として，それぞれの分析観点の重畳的な分析を行う．

Blocking Bug が特定の開発者に割り当てられている状況や特定の修正対象物が Blocking Bug を引き起こす可能性などについて調査することで，Blocking Bug の発生原因をより正確に捉えることができると期待される．

4.2. Blocking Bug の実態把握のための分析

- 不具合報告のメトリクス

1. **不具合の数**：Blocking Bug と非 Blocking Bug の不具合票の数を調査する．
2. **修正時間**:Blocking Bug，非 Blocking Bug，Blocked Bug の修正時間を調査する．本研究では不具合票の Status が “Resolved” または “Closed” のものを解決済みの不具合として扱うことにし，不具合が報告された日時から解決した日時までを修正時間と定義する．
3. **Blocking Bug の検出時間**：Blocking Bug が検出されるまでの時間を調査する．本研究では不具合が報告された日時から “blocks” のタグがつけられるまでの日時を Blocking Bug の検出時間と定義する．
4. **優先度別の不具合票の内訳**優先度別に Blocking Bug と非 Blocking Bug の不具合票の数を調査する．
5. **各 Blocking Set の構成不具合数**：各 Blocking Set が持つ不具合票の数を調査する．本研究では Blocking Set に含まれる Blocking Bug と Blocked Bug の合計を Blocking Set を構成している不具合票の数とする．
6. **Blocking Set の発生順序**：Blocking Bug と Blocked Bug のどちらが先に報告されたのかを調査する．Blocking Set の中でもっとも早く報告されたものが Blocking Bug である場合 Blocking Bug が先に報告されたものとし，1 つでも Blocked Bug が先に報告されていれば Blocked Bug が先に報告されたものとしてカウントする．

- 修正担当者のメトリクス

1. **各 Blocking Set の担当者の数**：各 Blocking Set に含まれるそれぞれの不具合の修正をおこなった修正担当者の数を調査する．
2. **修正担当者の経験**：Blocking Bug を埋め込んだ開発者の経験がどの程度であったかを調査する．開発者の経験はコミットの回数から測ることにする．まず，開発者を経験の浅い開発者と豊かな開発者に分けるために対象期間中の各開発者の総コミット回数を求める．対象期間中のコミット回数が第一四分位数以下である時期は「経験が浅い開発者」とし，第一四分位数以上である時期は「経験が豊かな開発者」であると定義する．

- 修正対象物のメトリクス：

1. **修正ファイル数**：Blocking Bug と非 Blocking Bug の修正ファイル数を調査する．本研究では不具合修正の際に変更されたファイルを修正ファイルとみなす．
2. **各 Blocking Set の修正ファイル数**：各 Blocking Set に含まれる不具合を修正するために変更したファイルの総数を調査する．

4.3. Blocking Bug の発生理解のための依存関係分析

本節では Blocking Bug の発生原因を調査するための取り組む RQ について述べる．不具合報告間の依存関係の観点から RQ1 を立てる．修正担当者の依存関係の観点からは RQ2 を，修正対象物の依存関係の観点から RQ3，分析視点間の依存関係の観点から RQ4 を立てる．

RQ1–1：優先度が高い Blocked Bug の発生によって Blocking Bug の修正は活発になるか

動機：報告されたすべての不具合をリリースまでに修正することは困難である．そのため各不具合には優先度が付与される．優先度は不具合が報告された時点で決定される（途中で変更することも可能）．不具合単体としては優先度が高いない不具合が Blocking Bug である場合や，Blocking Bug が重要でないとしても Blocked Bug が重要な可能性もある．単体では重要でないと判断され，修正作業を後回しにされた Blocking Bug によって，早期に修正しなければならない Blocked Bug の修正に取り掛かれないという状況が発生してしまう．今まで，修正作業が後回しにされていた Blocking Bug が重要であ

ることが確認されれば，不具合の優先度の割り当てをより適切なものにすることができる．結果，Blocking Bug の早期修正に役立つと考えられる．

アプローチ: 本研究では Blocked Bug の優先度が高く，修正作業が後回しにされた Blocking Bug が何件存在するかを確かめる．本研究では不具合票の Priority が “Blocker” または “Critical” のものを優先度が高い不具合とする．また，修正が後回しにされたかの判断については，不具合報告後，Blocking Bug の検出時間の中央値より長い時間，コメントやパッチ投稿を不具合管理システムにおこなわなかった不具合票を修正作業を後回しにされたと定義する．

RQ1–2 : 優先度が高い Blocked Bug の発生が Blocking Bug の発見に寄与するか

動機： RQ1–1 で述べた仮説では優先度の高い Blocked Bug の修正に取り掛かかることで Blocking Bug の存在が判明するケースも考えられる．RQ1–1 と合わせて，優先度の高い Blocked Bug の修正が Blocking Bug に発見に寄与するかどうかを RQ1–2 として調査する．

アプローチ： 優先度が高い不具合の定義は RQ1–1 と同様とする．報告された不具合票には不具合に関してのコメントや依存関係にある不具合の情報を追加登録（変更）することができる．Blocking Bug の不具合票において最初の追加登録（変更）が “blocks” タグをつける（Blocking Bug であると定める）ことであった場合を，優先度が高い Blocked Bug が Blocking Bug の発見に寄与したと考える．

RQ2 : 修正担当者が多いほうが Blocking Bug が発生しやすいか

動機 : OSS 開発現場では不特定多数の人々が開発に参加している．ファイルの変更が他のファイルへ影響する場合，修正担当者が複数人いることで変更されたファイルからの影響をよく理解しないまま修正をおこなう可能性がある．結果として Blocking Bug が発生する頻度が高まる．修正担当者が多いほうが Blocking Bug が発生しやすいということが確認できれば，不具合の修正担当者を割り当てる際に関連がありそうな不具合は 1 セットと考えて，1 人の担当者に任せることで Blocking Bug の発生を予防することが可能であると考えられる．

アプローチ： RQ2 に答えるために，修正担当者数別に Blocking Bug の件数を示す．修正担当者は Blocking Set の不具合の修正をおこなった開発者とする．

RQ3 : ファイルが参照されているほど Blocking Bug が発生しやすいか

動機 : 大規模なソフトウェアは複数のソフトウェア部品によって構成されている．複数のファイルを参照し，組み合わせることでソフトウェアを構築している．これはファイルごとに機能をまとめることでファイル管理の観点からは有効であると言える．しかし，ファイルを参照していることで予期しない不具合が発生することがある．例えば，機能追加等で他のファイルから参照されているファイルを変更するとする．このとき，ファイルの変更によって他のファイルに与える影響を十分に考慮する必要があるが．このように，ファイルの参照関係が多いほど同時に変更すべきファイルが多くなり，不具合が発生する可能性も高くなる．ファイルの被参照数が多いほど Blocking Bug になりやすいことが確認されれば，被参照数を減らすことで Blocking Bug の発生を予防することができる．**アプローチ :** RQ3 に答えるために，Blocking Bug と非 Blocking Bug のそれぞれのファイルが参照されているファイル数を比較する．比較には U 検定を用いる．参照されている状態について定義する．本研究ではインポートや継承によって他のファイルから利用されているとき，ファイルが「参照されている」と定義する．「参照されている」状態かを分析するためにはソースコード解析ツールである Understand[1] を用いる．

RQ4 : 経験が豊富な開発者によって Blocking Bug の発生を予防できるか

動機 : ファイルの変更権限が与えられた開発者（コミッタ）のコードやプロジェクトに対する理解は開発者ごとに異なる．経験が浅い開発者は経験が豊かな開発者に比べてファイルの依存関係の理解が浅い．ファイルの依存関係によって発生するような Blocking Bug は経験が浅い開発者のほうが発生させやすいと考えられる．経験の豊かな開発者がいる場合，こうした Blocking Bug の発生を予防することができる可能性がある．経験の豊かな開発者が Blocking Bug の発生を予防することが確認できれば，Blocking Bug が発生する可能性の高いファイルは経験豊富な開発者にチェック依頼することで発生を予防できると考えられる．

アプローチ : RQ4 に答えるためのアプローチについて述べる．経験豊かな開発者がいることで Blocking Bug の発生を予防できるかどうかを確かめるために，チームを作成する．Blocking Bug が埋め込まれたファイルに変

[1] Understand: https://www.techmatrix.co.jp/quality/understand/

更を加えた経験をもつ開発者群をチームとする．開発者の経験については 4.2 節の不具合担当者間のメトリクスで述べた定義を使うことにする．RQ4 に答えるために，経験豊かな開発者のみ，経験が豊かな開発者と浅い開発者の混合，経験の浅い開発者のみの 3 つのグループに分けて，各チームが発生させた Blocking Bug の件数を比較することにする．

5. ケーススタディ

5.1. 対象プロジェクト

本論文では，Apache Hadoop プロジェクトをケーススタディの対象とする．Hadoop プロジェクトは，大規模データの分散処理を可能とする OSS である．主に，Hadoop Common, HDFS, Hadoop YARN, Hadoop Map/Reduce の 4 つのコンポーネントで構成される．Hadoop を選定した理由は，4 つのコンポーネントが互いに密接な関係にあるため Blocking Bug が発生しやすいと考えたためである．

5.2. 対象データ

依存関係分析の対象データとして，不具合データ，ソースコード及びソースコードの変更履歴のデータを用いる．Hadoop プロジェクトでは，不具合データは不具合管理システム JIRA から，ソースコード及びソースコードの変更履歴は Git から取得することができる．対象データを作成するまでの 3 つの手順を以下に示す．

Step1：取得した不具合データから解決済みの不具合データのみを抽出する．解決済みの不具合データのみを抽出するのは，不具合の修正がおこなわれるまでは修正対象物（ソースファイル等）を特定することができないためである．本研究では不具合票の Resolution が “Fixed” のものを解決済みの不具合とする．**Step2**：Blocking Bug のうち Blocking Set を作成することができるものを抽出する．本研究では Blocking Bug の依存関係を分析したいと考えているため Blocking Set の構築が可能である Blocking Bug に限定する．

Step3：抽出した不具合データのうち，不具合データとコミット履歴の紐付けができるものを抽出する．コミットメッセージに不具合 ID が含まれていれば，該当する不具合を修正するためのコード変更を特定することができる．また，SZZ アルゴリズム [15] を併用することで，該当する不具合が混入したリビジョンを特定することができる．特定ができた Blocking Bug と関係のない不具合を本研究で扱う「非 Blocking Bug」とする．また，Blocking Set のすべての不具合が特定できたものを本研究で扱う「Blocking Bug」とする．

表 1. データセットの概要

コンポーネント	Blocking Bug	非 Blocking Bug
Hadoop Common	107	1,223
HDFS	57	1,851
Hadoop YARN	10	134
Hadoop Map/Reduce	60	1,575
全体	**234**	**4,783**

表 2. 修正時間

	Blocking Bug	非 Blocking Bug	Blocked Bug
修正時間（日）	10	11	52

5.3. Blocking Bug の実態把握のための分析

5.3.1 不具合報告のメトリクス

1. **不具合の数**

   Blocking Bug と非 Blocking Bug の不具合数についての分析結果を表 1 に示す．

   234 件の Blocking Bug と 4,783 件の非 Blocking Bug が確認された．

2. **修正時間**

   修正時間についての分析結果を表 2 に示す．

   表 2 では，Blocking Bug，非 Blocking Bug，Blocked Bug の修正時間の中央値を日数単位で示している．Blocking Bug と非 Blocking Bug の間で修正時間に大きな差はみられなかった（U 検定の結果，$p = 0.46$ であった）．また，Blocked Bug は修正時間が中央値で 52 日であることが確認された．

3. **優先度別の不具合票の内訳**

   優先度別に不具合票の内訳を表 3 に示す．

   高い優先度 (Blocker と Critical) が割り当てられた不具合の割合については，Blocking Bug と非 Blocking Bug の間で大きな違いはみられなかった．低い優先度（Minor や Trivial）が割り当てられた不具合の割合については，非 Blocking Bug に比べて Blocking Bug のほうが少ないことが確認された．

表 3. 優先度別の不具合票の内訳

優先度の割合	BlockingBug	非 BlockingBug
Blocker	25 (10.7%)	426 (8.9%)
Critical	12 (5.1%)	319 (6.7%)
Major	177 (75.6%)	2,965 (62.0%)
Minor	18 (7.7%)	854 (17.9%)
Trivial	2 (0.9%)	219 (4.6%)

表 4. 各 Blocking Set の基本統計量

	最大値	最小値	平均値	中央値	標準偏差
不具合票の数	4.0	2.0	2.2	2.0	0.2
修正担当者の数	3.0	1.0	1.4	1.0	0.3
修正ファイル数	9405.0	1.0	85.8	13.0	661.1

表 5. 構成不具合票数別の Blocking Set の件数

構成不具合票数	件数
2 件	187 (80.3%)
3 件	37 (15.9%)
4 件	9 (3.9%)

4. **Blocking Set の構成不具合数** Blocking Set の構成不具合数を表 4 の 1 行目に示す．表 4 に，Blocking Set の統計量（1 つの Blocking Set に含まれる不具合数，修正担当者数，修正ファイル数）を示す．Blocking Set に含まれる不具合の数の最小値と中央値が 2 件であることから，半数以上の Blocking Set は Blocking Bug と Blocked Bug が 1 件ずつであることがわかる．

   表 5 に Blocking Set の構成不具合数別の件数を示す．約 8 割の Blocking Set は，構成する不具合の数が 2 つである．つまり，約 8 割の Blocking Bug は 1 つの Blocked Bug の修正を妨害していることが確認された．

5. **Blocking Set の発生手順**

   Blocking Bug と Blocked Bug のどちらが先に報告されたかを表 6 に示す．

   Blocking Bug の約 7 割が Blocked Bug のほうが先に報告されていることが確認された．

5.3.2 修正担当者のメトリクス

1. **Blocking Set の担当者の数**

   Blocking Set に含まれる不具合を修正した担当者の数を表 4 の 2 行目に示す．修正担当者の中央値は 1 人であった．

表 6. 先に報告された不具合の数

	Blocking Bug	Blocked Bug
先に報告された不具合の数	69（29.5%）	165（70.5%）

表 7. 開発者の経験による Blocking Bug の発生件数

開発者の経験	件数	コミット回数
浅い	13	614
豊か	173	6066

表 8. 修正ファイル数（中央値）

	Blocking Bug	非 Blocking Bug
修正ファイル数	4	2

2. **修正担当者の経験**

   経験の違いによる Blocking Bug の発生件数とコミット回数を表 7 に示す．234 件の Blocking Bug のうち，新たなファイルの作成のみで修正を完了させた Blocking Bug が 48 件存在した．この 48 件の Blocking Bug を除いた，186 件の Blocking Bug に分析をおこなった結果，経験が浅い開発者は 47 回に 1 回の割合で Blocking Bug を発生させることが確認された．また経験が豊かな開発者は 35 回に 1 回の割合で Blocking Bug を発生させていた．

5.3.3 修正対象物のメトリクス

1. **修正ファイル数（中央値）**

   Blocking Bug と非 Blocking Bug の修正時に変更したファイルの数を表 8 に示す．分析の結果，修正ファイル数の中央値は Blocking Bug の方が多いことが確認された（U 検定をおこなった結果，$p < 0.00$）．つまり，Blocking Bug の修正には非 Blocking Bug と比べてより多くのファイルを修正する必要があることが確認された．

2. **各 Blocking Set のファイル数**

   各 Blocking Set に含まれる不具合を修正するために変更したファイルの総数を表 4 の 3 行目に示す．表 4 から修正総ファイルは最大で 9405 ファイル，最小で 1 ファイルであることが確認された．また中央値は 13 ファイルであった．

表 9. 修正担当者の人数の内訳

修正担当人数	件数
1	145（62.0%）
2	79（33.8%）
3	10（4.2%）

5.4. Blocking Bug の発生理解のための依存関係分析

RQ1–1：優先度が高い Blocked Bug の発生によって Blocking Bug の修正は活発になるか

243 件の Blocking Bug のうち，優先度が高い Blocked Bug を持っている不具合は 45 件確認された．45 件の Blocking Bug のうち，Blocking Bug の検出時間の中央値である 2 日を超えた後に修正が開始された Blocking Bug は 3 件のみであった．

RQ1–2：優先度が高い Blocked Bug の発生が Blocking Bug の発見に寄与するか

45 件の Blocking Bug のうち，不具合管理システムでの最初の変更が “blocks” のタグの挿入であるものは 4 件のみであった．

RQ1–1 と RQ1–2 の結果から優先度が高い Blocked Bug であっても Blocking Bug の発見にはつながらないことが確認された．

RQ2：修正担当者が多いほうが Blocking Bug が発生しやすいか

表 9 に Blocking Set の修正担当者の人数に関する詳細な内訳を示す．

全 Blocking Set のうち，約 62%が 1 人の担当者によって修正されていることが確認された．RQ2 の結果から，修正担当者が多くても Blocking Bug が発生するとは言えないことが確認された．

RQ3：ファイルが参照されているほど Blocking Bug が発生しやすいか

表 10 に Blocking Bug と非 Blocking Bug ごとの参照数の中央値を示す．

表 10 は Blocking Bug と非 Blocking Bug のそれぞれの参照数の中央値と，U 検定をおこなった際の p 値を示している．分析の結果，将来 Blocking Bug になるファイルのほうが，将来 Blocking Bug とならないファイルと比べてより多くのファイルから参照されていることが確認された．RQ3 の結果から，多くのファイルから呼ばれているほうが Blocking Bug になりやすいことが確認された．

表 10. 参照ファイル数（中央値）

	Blocking Bug	非 Blocking Bug	p 値
参照数	1	1	0.39
被参照数	4	3	0.00**

表 11. チーム別の Blocking Bug 発生件数

チーム	件数
経験豊富な開発者のみ	81
経験豊富な開発者と浅い開発者の混合	102
経験の浅い開発者のみ	3

RQ4：経験が豊富な開発者によって Blocking Bug の発生を予防できるか

開発者の経験を基に分けたグループの Blocking Bug の発生件数を表 11 に示す．

RQ4 の結果から，経験が豊富な開発者がいても Blocking Bug の発生を予防できるとは言えないことが確認された．

6. 考察

本章では，まずケーススタディの結果に基づいた考察をおこなう．その後，本研究の制約について述べる．

6.1. Blocking Bug とファイルの関係

本研究では，Blocking Bug が埋め込まれるファイルは他のファイルに比べて被参照ファイルが多いことが確認された．また，Blocking Bug は非 Blocking Bug に比べて修正するために変更しなければならないファイルが多いことも確認された．Blocking Bug が埋め込まれるファイルは参照されているファイルが多いので，ファイルの変更によって与える影響が大きく修正すべきファイルも多くなったと考えられる．また，これらの結果から，Blocking Bug は非 Blocking Bug に比べて修正にコストがかかると言えるので，早期修正と発生の予防の両方の観点から参照されているファイルを減らすことは重要であると考えられる．ファイルの被参照数を減らすための方法としては，メソッドの利用を継承に変更する方法や子クラスの共通メソッドを親クラスに引き上げるなどのリファクタリング方法が考えらえれる．

実際に Hadoop プロジェクトで修正されたファイルの依存関係の一部を用いて被参照ファイルを減らす方法を説明する．図 2 において “Writable” ファイルは Blocking Bug として修正されたファイルである．“Writable” ファイルは “Configuration” ファイルから継承のために，また，“WritableUtils” ファイルからはメソッドの利用のために参照されている．“WritableUtils” ファイルは “Configuration” ファイルからメソッド利用のために参照されている．このとき，“Configuration” ファイルは “Writable” ファイルを継承しているが，図 2 の下図のように “WritableUtils” ファイルが “Writable” ファイルを継承することで “Configuration” ファイルは “Writable” ファイルを継承する必要がなくなる．結果として，“Writable” ファイルは “Configuration” ファイルから参照されなくなり，参照されるファイルの数は 2 ファイルから 1 ファイルへ減少する．

6.2. Blocking Bug と修正担当者の関係

本研究では，約 62%の Blocking Set は 1 人で修正することが確認された．これは修正の効率化を重視した結果であると言える．例えば，修正担当者が複数人である場合，各修正担当者はそれぞれの不具合に関する情報を交わしながら担当する不具合を修正する必要がある．そのため，非 Blocking Bug に比べて修正に時間がかかる．しかし，修正担当者 1 人で Blocking Set に対応すれば，不具合に関する情報を交わす必要がなくなるので非 Blocking Bug 同様に議論のための時間は必要なくなり，修正が効率化されると考えられる．実際に，修正担当者が 1 人の場合と複数人の場合で Blocking Bug の修正時間の差を比較した結果，修正時間の中央値は，修正担当

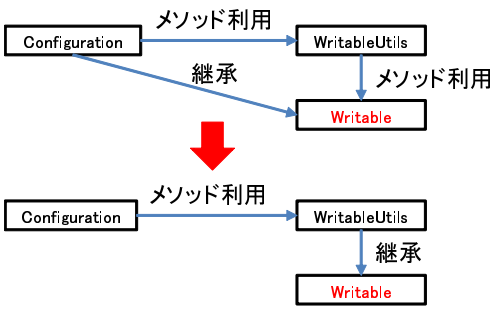

図 2. Hadoop プロジェクトの実態例

差者が 1 人である場合は 7 日であるのに対し，複数人である場合は 20 日であった（U 検定の結果 p 値 $= 0.02*$）．つまり，Blocking Bug の早期修正をおこなうためには Blocking Set を 1 人で修正する方が効率的であると言える．

6.3. 本研究の制約

本研究ではファイルを変更する際のコメントを用いて，不具合，モジュール，人の 3 つのデータを結びつけている．しかし，コメントの内容によっては結びつけができないこともあるため，結びつけができなかったデータは対象から除外している．また，不具合の解決方法によっては，結びつけができないために対象から除外しているものもある．これは，不具合の解決方法の中にはファイルの変更を伴わないものがあるためである．例えば，“Won’t Fix” とつけられている不具合表では，ファイルの変更がおこなわれる可能性は低い．よって，本研究で扱う Blocking Bug は Blocking Set 中のすべての不具合票で最低 1 ファイル以上の修正がおこなわれた Blocking Bug に限定される．

7. おわりに

本論文では Hadoop プロジェクトを対象に Blocking Bug の実態把握と発生原因を理解するための分析をおこなった．Blocking Bug の発生原因を理解するため，不具合間の依存関係，修正担当者間の依存関係，修正対象物の依存関係の 3 つの視点に加えて，3 つの視点の重畳的な依存関係の 4 つの視点から依存関係分析をおこなった．Hadoop プロジェクトを対象におこなったケーススタディの結果，Blocking Bug は Blocked Bug が先に報告されてから報告されることが多いことや，非 Blocking Bug に比べて修正のために変更を加えるファイル数が多いことが確認された．また，多くのファイルに参照されていると Blocking Bug になりやすいことが確認された．今後の課題としては，本研究で得られた知見を基に Blocking Bug の発生をどの程度予防できるかを確かめる必要があると考えられる．また，本論文で得られた知見は Blocking Bug 検出の予測やファイル変更時の影響分析にも活用できると考えられる．

参考文献

[1] Page-Jones, M.: *Practical Guide to Structured Systems Design (2nd Edition)* (1988).

[2] April, A. and Abran, A.: *Software Maintenance Management: Evaluation and Continuous Improvement* (2008).

[3] Shokripour, R., Anvik, J., Kasirun, Z. M. and Zamani, S.: Why So Complicated? Simple Term Filtering and Weighting for Location-based Bug Report Assignment Recommendation, in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR' 13)*, pp. 2–11 (2013).

[4] Menzies, T. and Marcus, A.: Automated severity assessment of software defect reports, in *24 th IEEE International Conference on Software Maintenance (ICSM '08)*, pp. 346–355 (2008).

[5] Tian, Y., Lo, D. and Sun, C.: DRONE: Predicting Priority of Reported Bugs by Multi-factor Analysis, in *29th IEEE International Conference on Software Maintenance (ICSM '13)*, pp. 200–209 (2013).

[6] Runeson, P., Alexandersson, M. and Nyholm, O.: Detection of Duplicate Defect Reports Using Natural Language Processing, in *Proceedings of the 29th international conference on Software Engineering (ICSE '07)*, pp. 499–510 (2007).

[7] Sun, C., Lo, D., Wang, X., Jiang, J. and Khoo, S.-C.: A discriminative model approach for accurate duplicate bug report retrieval, in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10) - Volume 1*, pp. 45–54 (2010).

[8] Kashiwa, Y., Yoshiyuki, H., Kukita, Y. and Ohira, M.: A Pilot Study of Diversity in High Impact Bugs, in *Proceedings of 30th International Conference on Software Maintenance and Evolution (ICSME2014)*, pp. 536–540 (2014).

[9] Chen, T., Nagappan, M., Shihab, E. and Hassan, A. E.: An Empirical Study of Dormant Bugs, in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR '14)*, pp. 82–91 (2014).

[10] Shihab, E., Mockus, A., Kamei, Y., Adams, B. and Hassan, A. E.: High-impact Defects: A Study of Breakage and Surprise Defects, in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pp. 300–310 (2011).

[11] Nistor, A., Jiang, T. and Tan, L.: Discovering, Reporting, and Fixing Performance Bugs, in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*, pp. 237–246 (2013).

[12] Garcia, H. V. and Shihab, E.: Characterizing and Predicting Blocking Bugs in Open Source Projects, in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR '14)*, pp. 72–81 (2014).

[13] Biazzini, M., Monperrus, M. and Baudry, B.: On Analyzing the Topology of Commit Histories in Decentralized Version Control Systems, in *Proceedings of the 30th International Conference on Software Maintenance and Evolution* (2014).

[14] Bhattacharya, P., Iliofotou, M., Neamtiu, I. and Faloutsos, M.: Graph-based Analysis and Prediction for Software Evolution, in *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pp. 419–429 (2012).

[15] Śliwerski, J., Zimmermann, T. and Zeller, A.: When Do Changes Induce Fixes?, in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, pp. 1–5 (2005).