SEC BOOKS

組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**情報処理推進機構**　技術本部 ソフトウェア・エンジニアリング・センター　編

はじめに

ソフトウェアの開発管理において、バグを速やかに、かつ確実に正しく修正し、その再発を防ぐことは製品品質を確保するために重要な活動です。そのためには一つひとつのバグをきちんと管理するとともに、バグ原因の傾向を分析し対策に繋げておくことが必要です。

近年、ソースコードの規模や連携する他のソフトウェアの増加により、プロジェクトの複雑化・大規模化が進みテスト工程で発見されるバグも増大し、バグ管理のための業務が煩雑になってきています。一方、オープンソースのバグ管理システムも手軽に利用できるようになり、多くのプロジェクトがバグの管理にツールを使用しています。しかし、単にツールがあるというだけではすべてのバグをきちんと管理して対策することが難しくなってきており、バグ管理を始めるに際して必要な管理項目の検討やバグ管理が円滑に進められるワークフローの検討に手間がかかる、実際の運用においてバグ管理担当者の選任と役割の明確化、各管理項目の入力が正しく実施されないなどといった課題が出てきています。

しかしながら、バグの管理について日本語でまとめられた情報や文献は少なく、その標準的なガイドとなるものが残念ながらほとんどないことも実態です。本書は、バグ管理に関するこのような課題を改善するため、日本の組込みシステム開発における標準的なバグ管理方法を示して普及させ、バグ管理の底上げやバグデータ測定の平準化を通して組込みソフトウェアの品質向上を図ることを目的として発行するものです。

この目的に沿うべく本書は、バグ管理の目的や方針が明確になること、またグローバル標準との整合を採ることを基本コンセプトとして、IPA/SEC のバグ管理手法部会で検討し編集しました。検討にあたって東洋大学経営学部の野中誠准教授に多くのコメントをいただきました。また、本書のバグカウントの指針は野中先生の論考を採り入れたものです。先生にお礼申し上げます。

本書を組込みソフトウェアの設計・テストを担当される方々をはじめ、品質管理に携わる多くの皆様方のバグ管理の活動にとって有用な情報として、ご活用いただければ幸いです。

2013 年春

独立行政法人 情報処理推進機構（IPA）
技術本部 ソフトウェア・エンジニアリング・センター（SEC）
組込み系プロジェクト
三原　幸博、十山　圭介
バグ管理手法部会
三橋　二彩子

目次

第１章

バグ管理と本書について

本章では、バグ管理の課題とその改善のための基本コンセプトを示し、バグ管理の目的や本書の構成、使い方を紹介します。

1.1 バグ管理における課題と本書の目的

近年、オープンソースのバグ管理システムも手軽に利用できるようになり、多くのプロジェクトがバグの管理にツールを使用しています。しかしながら、組込みソフトウェアの開発ではソースコードの規模や連携する他のソフトウェアの増加によってプロジェクトの関係者が増加し、テスト工程で発見されるバグも増加しています。このため、単にツールがあるというだけではすべてのバグをきちんと管理して対策することが難しくなってきており、以下のような課題が出てきています。

- **新規にバグ管理を始める場合、管理項目の検討やワークフローの検討に時間がかかる。**
- **実際の運用では、バグ管理の各項目の入力が正しく実施されない。**
  個々のバグ管理項目の目的が十分に理解されず、項目に対する入力データの内容が入力者によってばらつきがあり、正しく入力されないことがある。
- **バグデータの活用では、基本的な測定方法の指針がなくデータがばらつく。**
  例えばコーディング規約違反はバグかどうかなど、何をバグとしてカウントするのか。また、現れる現象は複数でもその原因が1カ所の場合に、バグ1件とカウントするかどうかなど。

また、バグの管理について日本語でまとめられた情報や文献は少なく、その標準的なガイドとなるものがほとんどありません。

本書はバグ管理に関するこのような課題を改善し、日本の組込みシステム開発における標準的なバグ管理方法を以下のコンセプトから示して普及させ、バグ管理の底上げやバグデータ測定の平準化を通して組込みソフトウェアの品質向上を図るガイドとなることを目的としています。

本書編纂の基本コンセプトは以下のようになります。

- **バグ管理の目的を示し、管理項目を整理して提示することで、適切な管理項目の設定とそれに対するデータ入力を促進できるようにする。**
- **昨今のグローバルな開発体制を考慮し、グローバル標準（IEEE や CMU/SEI**

など）との整合性を考慮する。
- 何をバグとしてカウントするかなど、バグ 1 件のカウントの仕方などの指針を示してバグカウントを平準化できるようにする。
- バグや障害（fault）、故障（failure）などバグに関連する用語を整理して提示し、それらの概念の理解を促進する。
- 実例をベースとして、組込みシステムのバグ管理における留意点やノウハウを提示する。

1.2 バグ管理の目的

バグ管理は、通常以下のような目的で行われます。

- バグの修正
- 製品リリース時に残存するバグの有無の把握
- バグの検出状況によるソフトウェア品質の推定
- バグの分析によるソフトウェア開発のカイゼン

バグ管理を行うことで、バグの発見から修正の割り振りや原因究明・修正・確認・承認など一連のバグ管理プロセスにより、多人数での組織開発を効果的に行うことができます。また、バグ管理の観点から対策漏れの防止や潜在バグの削減と市場流出防止、対策の効率化とスピードアップを行い、品質確保が図れます。

さらに、バグの傾向を明らかにすることでコーディング規約や設計法へフィードバックができ、課題抽出と改善のための PDCA にバグ管理を繋ぐことにより、組織としての開発能力向上と品質向上を図ることもできます。

適切なバグ管理を行うことによって、以下の効果も期待できます。

- 発見から解決まで、すべてのライフサイクルを通じたバグ管理ができ、バグが未解決のまま残ったり早期修正が必要なバグを見落としたりすることが無くなる。
- 開発プロジェクト全員が最新の状況に基づいて意思疎通ができ、解決したバグ情報をプロジェクトで共有できる。
- 開発者は、プロジェクトや顧客のそれぞれにとって重要な観点から修正を

進めることができるようになる。

●管理項目をデータベース化し統一して扱うことで、バグの追跡が自動化され、正確な分析ができるようになる。

1.3 本書の使い方と構成

●本書が対象とする領域

バグ管理には、対象製品の開発中に行う管理とリリース後に行う管理がありますが、本書で採り挙げる管理のスコープは開発中のものとしています。また、バグと考えられる問題は、ソフトウェア開発のV字モデルの図（図1.1）で、ソフトウェア要求定義やソフトウェアアーキテクチャ設計などの左側のプロセスでも発見されますが、本書において管理の対象とするものは、この図の実装より右側のプロセスで発見されるものとしています。

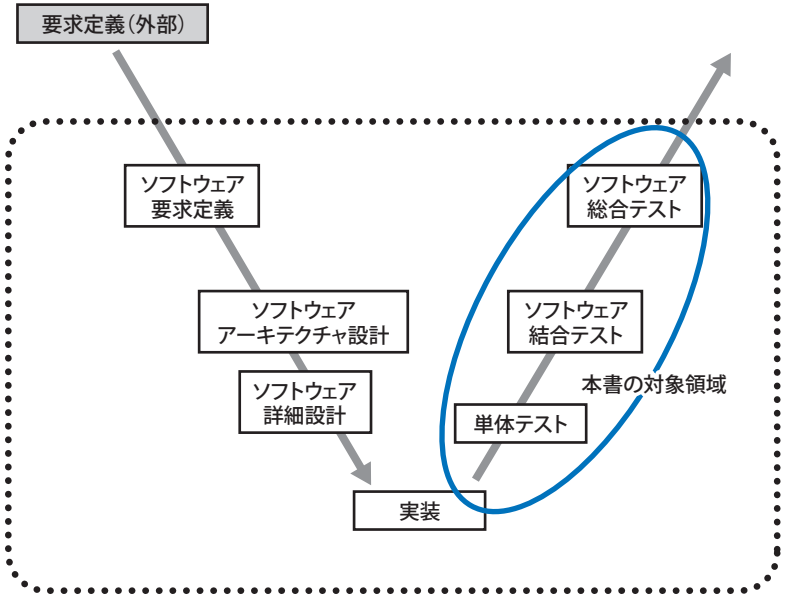


図1.1　ソフトウェア開発のV字モデルと本書の対象領域

●本書の対象者

本書は、組込みソフトウェアの開発における以下の方々にご利用いただくことを想定しています。

（1）バグ管理環境を整備する品質管理推進者やプロジェクトマネージャ
（2）バグ情報を入力するテスト実施者、開発者

●本書によって期待される効果

本書を参考にして、新規にバグ管理プロセスを実現する場合に短期間で適切な管理項目を決めることができ、その利用法や製品リリースまでのプロセスも明確にできます。また、管理項目や運用における入力について、なぜそうするかが理解できるようになり、データ入力の均質化が進み、示されたバグ測定の指針によって、標準的な測定ができるようになります。

●本書の構成

第2章でバグにまつわる用語について標準規格などを参考にして紹介します。その後、第3章でバグ管理プロセスについて、第4章で管理のために必要となる標準的な項目とバグの分類について説明します。第5章ではバグのカウントの仕方について指針を示し、第6章で事例をもとにバグの分析について説明します。

また、大量のバグ報告の扱いやバグ管理のヒント、バグの収束に関してなどをコラムとして関連箇所に示しています。

●本書のスコープとESxRとの関係

IPA/SECでは組込みソフトウェアの開発のために、開発プロセスの標準的な考え方を整理したESPR、プロジェクト計画書の立案方法を紹介するESMR、コーディング規約作成のためのESCR、品質作り込みのための指標を説明するESQR、品質を担保するための設計作法の事例集ESDR、ソフトウェアテストに関する考え方や知見を整理した「組込みソフトウェア開発における品質向上の勧め〔テスト編～事例集～〕」をESxRシリーズとして発刊してきました。バグ管理を中心にこれらと本書との関係を以下でまとめ、図1.2で示します。

（1）ESPRとの関係

バグ管理のためのアクティビティを詳細化する際の参考情報となります。また、システムテストやソフトウェアテストにおける準備・確認、プロセス全体の改善への入力とすることができます。

（2）ESMR との関係

バグ管理のための内部体制や人員計画、品質保証計画をプロジェクト計画書に織り込む際の参考情報となります。特に、品質保証の進め方や主要なイベントを明確にすることができます。

（3）ESCR との関係

バグの管理によって「よくあるバグ」が明らかになり、それを防ぐためのコーディング規約制定の検討に活かすことができます。

（4）ESQR との関係

ESQR の品質指標の一つである「バグ」のカウント方法の考え方を示しています。これによって、品質評価指標としての使い方と関連付けることができます。

（5）ESDR との関係

改善作業をプロジェクトに展開する中で、設計や改善のパターンとして設計作法に活かすことができます。

（6）「組込みソフトウェア開発における品質向上の勧め〔テスト編〜事例集〜〕」との関係

バグ管理とテストには密接な関連がありますが、バグの検出と修正を効率的にするテスト資産の蓄積方針やテスト設計、テストのスケジューリング、終了条件、ツールの活用などと連携させることが重要です。

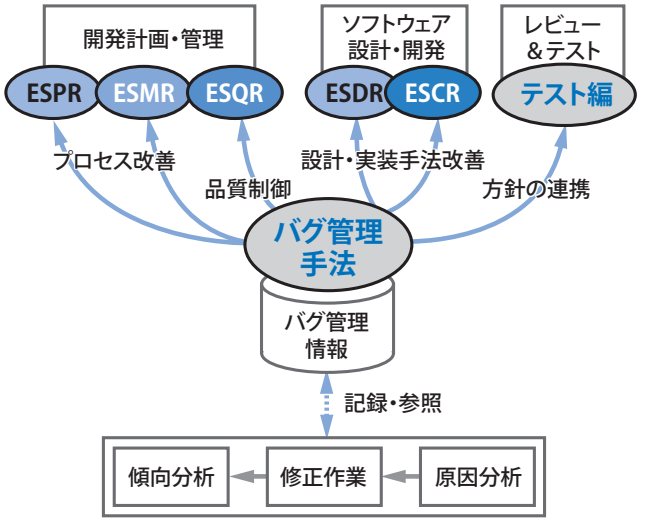


図 1.2　バグ管理手法と ESxR との関係

第２章

バグに関連する用語について

本章では、バグ管理を行う上で理解しておくべき用語について説明しています。JIS規格間でも同じ用語に対する意味が若干異なっており、使用する組織や個人でも受けとる意味合いにずれがあります。

2.1 「バグ」とは

普段使っている「バグ」という言葉の意味を考えたことがあるでしょうか？「バグ」とはシステムで表面化した動作不良の現象でしょうか？　それとも原因でしょうか？

我々はプログラムが望んだ通りの動作をしない場合、「バグった」「障害が発生した」「故障した」など様々な表現をします。残念ながら現状では、これらの言葉の定義は定まっていません。

我々が普段意識せずに使っている「バグ」に類する言葉をリストアップしてみると、以下のような言葉が出てきます。

バグ	不具合	欠陥	障害	故障	エラー

これらを「広辞苑（第六版）」で調べてみると、以下のように説明されています。

【バ　グ】（虫の意）コンピューターのプログラムの誤り・欠陥。
【不具合】製品などの、具合が良くないこと。多く、製造者の側から、「欠陥」の語を避けていう。
【欠　陥】かけて足りないもの。不足。不備。欠点。
【障　害】①さわり。さまたげ。じゃま。
【故　障】①事物の正常な働きがそこなわれること。さしさわり。さしつかえ
【エラー】①誤り。過失。③誤差

では、英語ではどうでしょうか。英語でも様々な表現があるようですが、やはり同じ単語でも別の意味で使われていることが多いようです。「研究社新英和大辞典（第六版）」で調べてみると、以下のように和訳されています。

【bug】　3.《口語》（機械の）故障、（計画などの思いがけない）欠陥（defect）、（コンピュータープログラムの）誤り、バグ
【defect】　1. 欠陥、欠乏、不足　2. 欠点、短所、弱点
【fault】　3. 誤り、過失、過誤　5.《電気》（回路の）障害、漏電
【failure】　4. 不足、欠乏、不十分　6 機能［運転］停止
【error】　1. 誤り、間違い　2a. 誤信、思い違い、心得違い　4.《数学・統計》（計算・観測などの）誤り、誤差

これら辞書の説明を見ても、すっきりとはしないようです。辞書の解釈も、人によっては違和感を覚える場合がありそうです。

本書の目的である「バグ管理」を行うにあたり、本章では共通の認識のために「バグ」「不具合」「障害」などの様々な呼び方のある事象について用語を整理します。

2.2 標準規格などによる定義

本書で用語を整理するにあたり、JIS などの標準規格やその他の業界標準における定義を見ておきます。表 2.1 にそれらの標準による定義の一覧を示します。出典によっては定義の無い言葉や、定義があってもそれぞれで微妙に内容が異なっていたりします。

表 2.1 用語定義一覧

出典	障害 (fault)	フォールト (fault)	故障 (failure)	欠陥 (defect)
JIS X 0133-1:1999 ソフトウェア製品の評価	計算機プログラム内の不正確なステップ、プロセスまたはデータの定義	(定義無し)	要求された機能を遂行する製品の能力が尽きる状態、または事前に仕様化された制限内での機能を遂行する能力が無い状態	(定義無し)
JIS X 0014:1999 情報処理用語ー信頼性、保守性及び可用性	要求された機能を機能単位が遂行できなくなる偶発的条件	(定義無し)	要求された機能を遂行する、機能単位の能力が無くなること	(定義無し)
JIS Q 9000:2006 品質マネジメントシステムー基本及び用語	(定義無し)	(定義無し)	(定義無し)	意図された用途または規定された用途に関連する要求事項を満たしていないこと
JIS C 0508-2:2012 電気・電子・プログラマブル電子化安全関係の機能安全	(定義無し)	機能ユニットに要求される機能遂行能力の低下または損失を引き起こすであろう異常状態	ある機能ユニットの要求される機能遂行能力の終結	(定義無し)

出典	障害（fault）	フォールト（fault）	故障（failure）	欠陥（defect）
JIS Z 8115:2000 ディペンダビリティ（信頼性）用語	（定義無し）	ある要求された機能を遂行不可能なアイテムの状態、また、その状態にあるアイテムの部分※	アイテムが要求機能達成能力を失うこと	（定義無し）
ソフトウェアテスト標準用語集 V 2.0（日本語版）	defect参照	（定義無し）	コンポーネントやシステムが、期待した機能、サービス、結果を提供できないこと	要求された機能をコンポーネントまたはシステムに果たせなくする、コンポーネントまたはシステム中の不備。例えば、不正な命令またはデータ定義。実行中に欠陥に遭遇した場合、コンポーネントまたはシステムの故障を引き起こす。
IEEE Std 1044-2009（バグ管理手法部会訳）	ソフトウェアにおけるエラー（error）の現れ	（定義無し）	・要求された機能を遂行するための製品の能力が終了すること、または前もって指定された制限内で要求された機能を果たせないこと ・システムまたはシステムのコンポーネントが指定された制限内で要求された機能を果たさない事象	プロジェクトの構成要素において、要求や仕様に合致せず修復か取り替えが必要であり、不完全であること、または不足していること
IEEE Std 982.1-2005（バグ管理手法部会訳）	コンピュータプログラムにおける正しくないステップまたは処理、データ定義（IEEE 610.12-1990 fault(2)の引用）	（定義無し）	システムまたはコンポーネントが指定された性能要求内で要求された機能を果たせないこと（IEEE 610.12-1990 failureの引用）	原因であるfault、または結果であるfailureのいずれも言及できる総称（Hand book of Software Reliability Enginerring, R.Lyu, 1996の引用）

※備考 3. に「ソフトウェアアイテムの場合、コンピュータシステムではプログラム全体でのプログラミング、論理構成、プログラムプロセス、データ定義などの間違いを意味する」として、ソフトウェアの場合の考え方が示されています。

本書では、JIS X 0133 や IEEE 1044 の定義に沿って「障害（fault）」「故障（failure）」という用語を用い、「障害」が原因となって「故障」が引き起されるものとします（図 2.1）。

ソフトウェアについて考えると、プログラムモジュールの「故障」の原因の例として、そのモジュールに内在する機能的な不備や不具合、例えば 0 除算を回避するなどの処理が正しくなされていないことやデータ定義の間違いなどが挙げられます。これが「障害」に対応しています。

図 2.1　「障害」「故障」の関係

2.3 本書における「バグ」の定義

ソフトウェアに関しては「ソフトウェアが故障した」という表現はあまり使われず、「不具合」や「欠陥」、「バグ」などの言葉が一般的です。いわゆるソフトウェアのバグは「障害（fault）」で、それが原因でソフトウェアが意図した通りに機能しない現象が「故障（failure）」であると捉えることができます。ただ、「バグ」が「故障」の原因である「障害」を指すのか、それとも「故障」という現象自体を指すのかについては、意見が分かれるところです。

本書では、「バグ」という用語を以下のように、不具合の原因とシステムの不具合自体の両者を指す言葉として使うことにします。

【バグ】設計者の認識の有無にかかわらず、すべての成果物において要件定義の誤り、仕様設計の誤り、プログラミングの誤り、システム構築の誤りなどにより「期待される結果」と乖離があるために、何かしらの対策・対応が必要と考えられる現象またはその原因。
特に現象と原因を区別する場合は、現象を示すにはバグ現象、原因を示すにはバグ原因とそれぞれ表記する。

この「バグ」の定義は IEEE 982.1-2005 の defect（欠陥）に対応しています。

●標準規格における「バグ」という言葉の定義について

「バグ」という用語は標準規格では定義されていません。JIS Z 8115 の解説では、バグという言葉の規格への採用について、以下のように説明されています。

「JIS Z 8115　ディペンダビリティ（信頼性）用語解説」（関連解説 16）より IEC 60050（191） 改正案中の 03-07：バグについて 現在 IEC 60050（191）改正案として 1/1693A/CDV が配布されている。その中では 191-03007 bug: A state of software characterized by a design fault とされている。

今回の JIS 改正では この用語の採用を見送り、解説で補うことにした。

（中略）

今日、製品の不具合、特に機能的な不備や欠陥を指して“バグ”ということがあるが、ソフトウェアの欠陥に関する総合的な概念を指す言葉が一般化したものと思われる。

（中略）

バグには以下のような 3 種類の意味が込められているとする意見もある。すなわち“人間の思考過程における人の誤りに代表される不完全性又は過失”、または“そのような過失の結果として設計に表現されてしまった記述上の不十分性、不統一性、不完全性、矛盾性のような欠陥”と“そのような記述上の欠陥が原因となって、ソフトウェアが稼動している時に外部から観測される機能、性能、使い易さ、保守のし易さなどの不具合”である。

（後略）

この解説にもあるように「バグ」という言葉は、欠陥そのものと、欠陥が原因となって起こるシステムの不具合の両者を指すことがあるという考え方が一般的なようです。

「バグ」と同様に「障害」という言葉も、一般的な日本語としてはシステムの不具合を起こす「妨げ」、例えばサブシステムの欠陥などシステムの不具合の原因と、その不具合自体の両者を指すことがあるようです。システムが機能しない状態（定義としては「故障」）を利用者から見て「障害が発生している」と表現する場合もあります。

用語が多義になっていては規格をきちんと制定することができないため、標準規格では用語の定義を一意に定めていますが、このような事情が規格の用語と一般の言葉の使い方との間に違和感を生み出す原因になることもあるようです。

Column

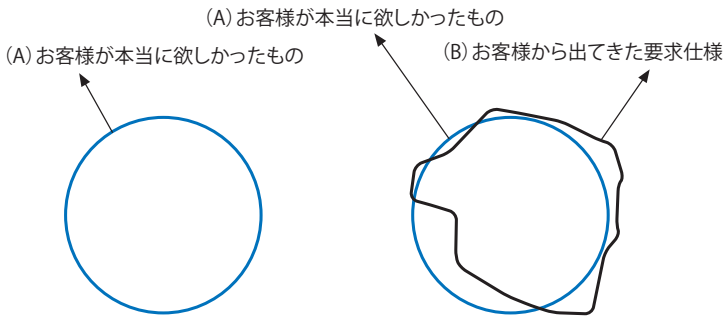
どこからバグと呼ぶのでしょうか？

バグ管理を実施するにあたり、指摘されたバグを「非バグ」として対策を実施しない場合があります。そのときの理由として、「仕様通り」「仕様変更」というケースが多いのではないでしょうか？　では、どこから「仕様通り」「仕様変更」と呼ぶのでしょうか？

ソフトウェア開発の現場では、以下のようなことが起きていると考えられます。

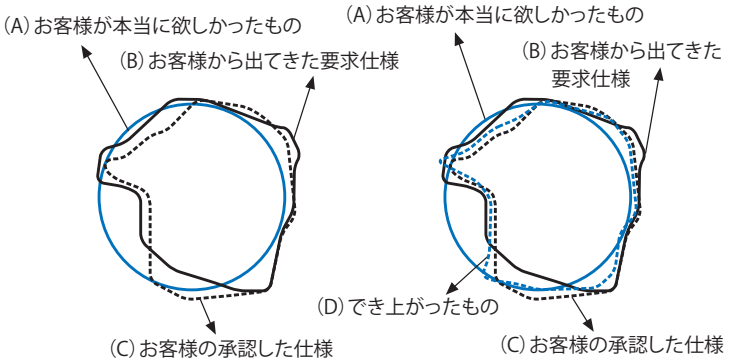
（1）ソフトウェアを開発するにあたり、お客様が本当に欲しかった仕様というものがあります。

（2）しかしながら、お客様の出す要求仕様は、本当に欲しかったものを正確に表現できていない場合があります。



（3）開発者はお客様から出てきた要求を仕様化して仕様書を作成します。そしてこの仕様書の内容についてお客様から承認をいただきます。ここでも、お客様の要求を正しく仕様化できていない場合、さらに差が広がります。

（4）仕様書通りに開発を実施しますが、その成果物も、残念ながら仕様書通りにできていない場合もあります。気がつけば、(A)と(D)の差がかなり広がってしまいました。



不幸にして、「お客様が本当に欲しかったもの」と「でき上がったもの」との間にかなりの乖離が発生してしまいました。お客様に満足していただけるのでしょうか？

ところで、バグとは(A) (B) (C) (D)、どれとどれの差をいうのでしょうか？

通常、「(C) お客様の承認した仕様」を正として、「(D) でき上がったもの」との違いをバグとして対策することが一般的だと思います。しかしながら、お客様から見た場合、費用をかけて完成したものが「(A) お客様が本当に欲しかったもの」通りにできていないことは納得できないものであり、受入れテストではこれもバグとして指摘されたりします。

また、お客様によっては、「要求を仕様書にブレークダウンするときの差 (C) − (B) は仕様書を作成した開発側の問題で、バグである」と主張する方がいるかもしれません。それに対して開発側は指摘内容が (C) と違っているのでバグではないとして扱い、「仕様変更」として修正を入れる場合は、「有償対応」と値増し交渉に入ったりします。

本書では、指摘されたものはすべてバグ管理対象とします。その中で、開発のベースラインと異なる指摘はバグではないものとして扱うように考えてあります。一般的には「(C) お客様の承認した仕様」をベースラインとして、バグか否か判断をしているようです。

お客様の本当に欲しかったものと同じものができ上がれば、このような悩みは発生しなくなります。(A) (B) (C) (D) それぞれの差分をいかに小さくしていくかは、永遠のテーマでもあります。このようなプロジェクトでは (B) (C) の差が大きいため、プロジェクト後半に手戻り作業が発生し混乱するケースも見られます。本当にお客様が欲しいものを仕様化する技術が大切であるといえます。

第３章

バグ管理プロセス

バグが発見されてから、分析や処置が行われ、対応が完了したことが確認されるまでの一連の活動のことをバグ管理プロセスといいます。

バグ管理プロセスを実現するために、データベースシステムを利用したバグ管理システムなどを運用し、その管理データを利用することで、統計的なバグ分析なども行われています。

3.1 いかにバグを管理するか

開発によっては、様々な要因により、多数のバグが発生します。バグを漏らさず適切に処置し、再発を防ぐためには、バグに関する情報を記録し、管理する必要があります。

バグの管理では、開発者や関係者が適切に対処できるよう、開発組織内での一連の活動として、バグ管理プロセスを定義することが一般的です。バグが発見されてから、分析や処置が行われ、対応が完了したことが確認されるまでの一連の活動のことをバグ管理プロセスといいます。また、バグに着目した表現として、発見から問題解決まで、バグの処置状態の変遷をバグのライフサイクルといいます。

バグ管理プロセスを実現するために、データベースシステムを利用したバグ管理システムなどを運用し、その管理データを利用することで、統計的なバグ分析なども行われています。

3.2 開発工程で発見されるバグ

リリースまでの開発工程で発見されるバグは工程内バグと呼ばれることがあり、開発プロジェクトデータの一つとして管理します。

工程内バグには、レビューなどで指摘される仕様書類の設計不具合やコーディング時に発見されるコード不具合、及びテスト工程で検出される不具合などがあります。

本章では、工程内バグを対象とした管理プロセスを解説します。リリース後のバグは「出荷後バグ」などと呼ばれますが、ここでは取り扱いません。

また、それぞれの現場では、「工程内バグ」「出荷後バグ」という用語ではなく、「流出バグ」など、違った用語が使われることもあるでしょう。

3.3 バグ管理プロセスの基本フロー

以下、図 3.1 に、一つのバグが報告され、対応が完了するまでの基本的な流れを示します。バグと考えられる不具合の内容や処理状況を管理するための記録をバグ票（バグ管理票、チケット）と呼んでいます。

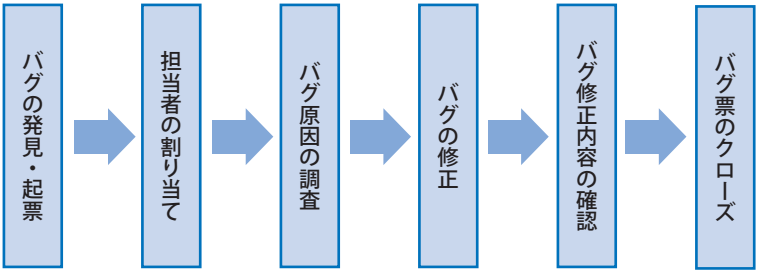


図 3.1　バグ対応の基本的な流れ

（1）バグの発見・起票

発見されたバグは、バグ管理システムや帳票などに報告されます。報告の完了時にバグ票の状態は「起票済」となり、管理者などの関係者に通知されます。

（2）担当者の割り当て

起票された情報を確認し、適切な担当者を割り当てます。担当者が割り当てられるとバグ票の状態は「担当者割当済」となり、担当者に通知されます。

（3）バグ原因の調査

担当者は再現性の確認、バグの原因調査、修正方法の検討などを行います。調査後、解決方法などの情報を合わせて記録し、バグ票の状態を「調査済」とします。

この調査で、バグではないと判断される場合もあります。その場合、バグ票に問題がないことを明記し、調査記録などを残しておきます。その後、「(6) バグ票のクローズ」処理に移ります。

（4）バグの修正

担当者は実際の修正作業を行います。バグを修正後、バグ票の状態を「処置済」とします。報告は確認担当者などに通知されます。

（5）バグ修正内容の確認

担当者は再テストを行い、修正が完了していることを確認した上で、バグ票

の状態を「検証済」とします。

（6）バグ票のクローズ

管理者は「検証済」となっているバグに対して内容を確認し、バグ票の状態を「完了」に変更します。

3.4 バグ管理の状態遷移

基本フローに従ってバグ管理システムを使用してプロセスを実現するときの実際を紹介します（図 3.2）。ここでは、バグ票のことをチケットとしています。

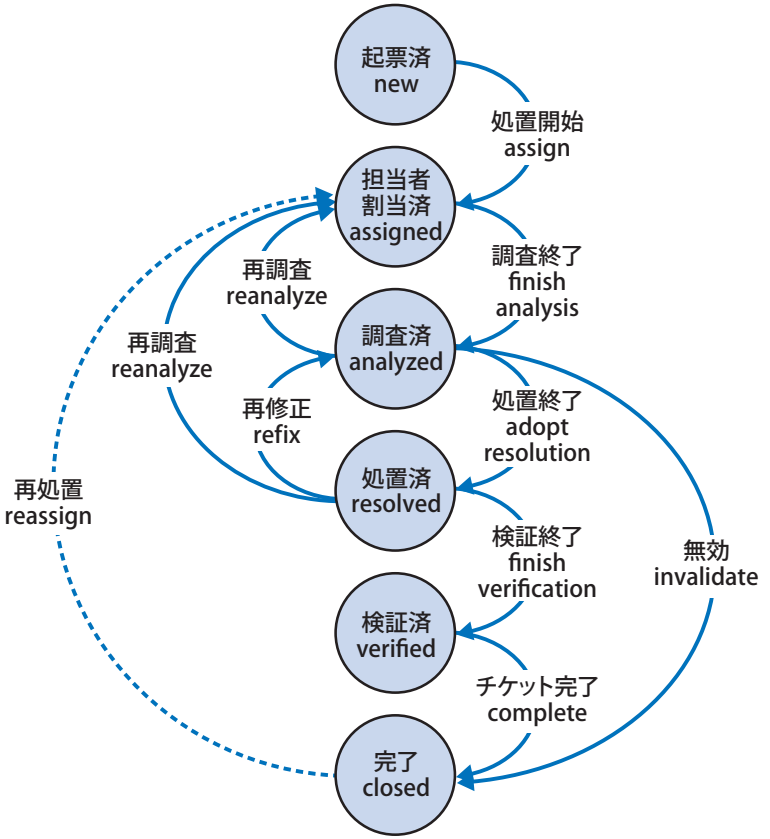


図 3.2　チケット（バグ票）の状態遷移

図中の円はチケットの状態を示し、バグのライフサイクルに対応します。矢印は各状態間の遷移を引き起こすイベントを表しています。

以下、表 3.1 に各チケットの状態を解説します。

表 3.1　チケット（バグ票）の状態

状態	意味
起票済 new	・バグ管理システムにチケットが投稿された状態。 ・まだ、チケットに担当者は割り振られていない。
担当者割当済 assigned	・担当者が割り振られた状態。 ・原因、対応などの調査活動を開始する。
調査済 analyzed	・バグの再現性が確認され、原因が調査され、修正案が検討された状態。 ・修正処置を開始する。
処置済 resolved	・修正案に従い、処置が終了した状態。 ・修正内容の検証を開始する。
検証済 verified	・ピアレビューなどが行われ、修正内容の検証が完了した状態。 ・チケットの最終確認を開始する。
完了 closed	・チケットがクローズされた状態。 ・これ以上の対応は行われない（再処置となる場合もある）。

状態遷移ごとに、チケットにどのような補足情報を入力するかは、バグ管理システムの仕様をもとに、開発組織でカスタマイズした入力ルールとして定められている場合が一般的です。実際の例や解説は、「第 4 章 4.1 バグ管理項目の一覧」を参照してください。

以下、表 3.2 に図 3.2 の各矢印が示すイベント内容を解説します。

表 3.2　状態遷移のイベント内容

活動	内容	結果と遷移先
処置開始 assign	・バグの内容を確認し、適切な担当者を割り振る。 ・バグが発見された環境やバージョン情報など、発見時の情報が記録されていることを確認する。	担当者割当済(assigned)に遷移する。調査が開始される。
調査終了 finish analysis	・担当者がバグ内容を確認し、分析作業を開始する。 ・分析結果から処置方法を検討する。 ・代替案を含めた処置方法のリスト、検討結果を記録する。 ・処置期限や対応予定バージョンなどの情報を入力する。	調査が完了したら、調査済(analyzed)に遷移する。
処置終了 adopt resolution	・担当者が分析結果を確認し、修正作業を実行する。 ・修正作業のレビューを行う。 ・処置内容やレビュー結果を記録する。	修正が完了したら、処置済(resolved)に遷移する。
検証終了 finish verification	・処置内容の検証を開始する。 ・レビュー記録や確証情報の参照先などを記録する。	検証が完了した時点で、検証済(verified)に遷移する。
無効 invalidate	担当者がバグ内容を確認し、分析した結果、バグではないと判断された場合。	バグではないと判断された時点で、完了(closed)に遷移する。
チケット完了 complete	・チケットの内容を確認する。 ・レビュー記録や確証情報の参照先などを記録する。	選択された処置内容が適切で、問題が解決されたと判断されたら、完了(closed)に遷移する。
再調査 reanalyze	・分析結果、処置内容を確認して、再調査が必要だと考えた場合。 ・再調査理由を明記する。	分析調査、修正方法が適切ではない場合は、担当者割当済(assigned)に遷移する。
再修正 refix	・処置内容を確認して、再修正が必要だと考えた場合。 ・再修正理由を明記する。	処置内容が適切ではなく、問題が解決されない場合は、調査済(analyzed)に遷移する。
再処置 reassign	リグレッション(バグの再発)が発見された場合など、このバグを再度取り扱う状態になったとき。	担当者割当済(assigned)に遷移する。

リグレッション（バグの再発）が発見された場合など、過去に発見されていたバグを再度取り扱う必要が出てきた場合、既存のチケットを再処置（reassign）にするか新しくチケットを発行して起票済（new）にするかは、バグを管理する目的や後のデータ分析への利用目的などによって異なるため、様々なルールが考えられます。

また、顧客などのステークホルダー（利害関係者）がいる場合、実行される検証のステップに合わせて検証済（verified）状態が複数個設定される場合があります。

適応するプロジェクトの規模・性質に合わせて、バグ管理プロセスをカスタマイズすることが重要です。

Column

大規模開発における留意点 その1
未解決バグは定期的に棚卸する

発見されてから一定時間経過しても解決されないで残っているバグは、いったんクローズします。もちろん原因不明・対策無しという記録は残しておきます。棚卸する理由は、以下のものが挙げられます。

（1）現行のソフトウェアのバージョン（構成）とバグが発見されたときのバージョン（構成）が異なっており、現象を再現することが困難であるため。
（2）現行のソフトウェアそのものがすでに仕様変更、修正により原因究明が困難になっているため。
（3）発見時のソフトウェアで原因が推定されたとしても、既に当該ソフトウェアの仕様が変化していたり、別の修正が入っていたりすることがあるので対策が難しいため。
（4）原因が推定されて修正した場合、他の部分への影響を確認することが難しいため。
（5）原因が推定されて修正した場合、確認のためのテストが難しいため。

このような状況で放置しておくと、担当エンジニアのリソースがロックされてしまい、他のバグ解決が遅れていくという不都合が発生する危険があります。また、いつまでたっても品質不良の状態が続くことになります。そのため、いったん棚卸してしまい、その後に改めて同じ現象のバグが発見された場合には、新たなバグとして取り扱うのがよいでしょう。

棚卸の周期は、システムのバージョンが2世代以上異なるまでの期間を目安とすることが多いようです。

3.5 バグの原因調査と修正作業の実施

基本フローで示した「バグ原因の調査」「バグの修正」を、開発現場で実施するときには、図 3.3 に示すような詳細な活動が含まれます。これらの活動に付随する情報を管理項目としてバグ管理システムに記録し、共有できる仕組みを作ることが重要です。

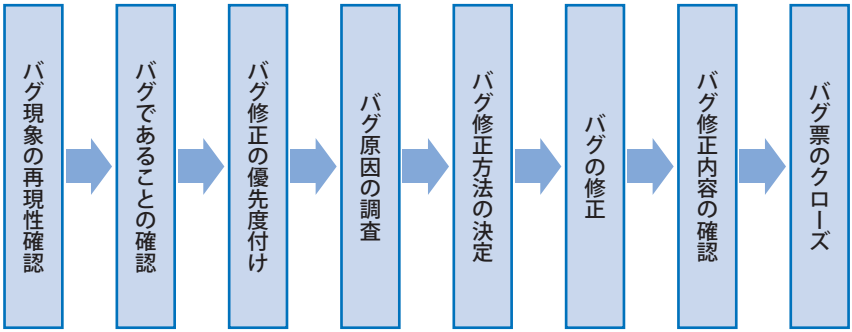


図 3.3　バグ対応の詳細な流れ

（1）バグ現象の再現性確認

発見された現象が再現するかどうか確認します。再現性が確認された場合は、次の活動に移ります。再現性が確認できない場合は、バグ票を通じて発見者に再度内容を問い合わせるなど、報告内容の再確認を行います。

（2）バグであることの確認

発見された現象が本当にバグであるかどうかを確認します。発見者の知識不足、確認不足、勘違いなどにより、発見された現象が、設計仕様通りの結果である場合もあります。この場合、バグ票は無効として扱い、完了処理が行われます。

（3）バグ修正の優先度付け

バグであることが確認できた後、そのバグの今後の扱いに対して優先順位を決定します。重大なバグやすぐに対処すべきバグである場合は優先順位を高く設定し、そうでない場合は低く設定します。

（4）バグ原因の調査

バグが発生した原因の調査を行います。この調査では、バグの発生場所を特定し、発生した原因を分析、その原因が起こった工程を特定します。

調査した結果は、バグ票などに記載します。調査結果を別の書面にまとめ、バグ票には、まとめたドキュメントのリンク先を記載する場合もあります。

（5）バグ修正方法の決定

バグ原因の調査結果から、バグの修正方法を決定します。修正方法が複数ある場合、その修正を選択、実施することによる製品やプロジェクトに対する影響を考慮する必要があります。また優先順位や、その方法が必要とする工数などから、状況に応じて適切な方法を選択します。

検討、決定した結果は、バグ票などに記載します。別の書面にまとめ、バグ票には、まとめたドキュメントのリンク先を記載する場合もあります。

（6）バグの修正

上記「（5）バグ修正方法の決定」に従ってバグの修正を行います。この時、決定した以外の修正をしないように注意してください。決定したこと以外の修正を加えると、新たなバグを発生させる原因になる場合があります。

（7）バグ修正内容の確認

修正されたバグが決定した修正方法で正しく行われているか、また、修正の影響範囲が想定した範囲内に収まっているかを確認します。

うまくバグが修正できなかったり問題が新たに発見されたりする場合は、再調査、再検討を行います。

（8）バグ票のクローズ

すべての必要事項が記入されていることを確認し、記入されていない場合は記入して、バグ票をクローズして、一連のバグ修正プロセスを完了します。

Column

大規模開発における留意点 その2
類似バグを一掃する

大規模開発においては、バグの発見・修正作業をいかに早く進めるかが、プロジェクトの成否を決めると言っても過言ではありません。もちろん製品テストまでにできるだけ品質を上げておくことは必須ですが、規模が大きいと内在するバグの数もどうしても大きくなりがちです。

そこで、早くバグの数を減らす方法の一つとして、類似バグを一掃するというものがあります。これは、バグが発見されて原因が究明された時に、それ一つだけを直すのではなく同時に類似のバグを見つけてまとめて直してしまおうというものです。バグの原因が仕様の取り違いであった場合、他の場所でもほぼ同じ間違いを起こしていると考えられます。閾値の誤りや不等式の誤り、設定値の誤りなどがその例になります。また、アルゴリズムや計算式、計算順序の誤りなどもほとんどの場合、同じコーディングパターンの間違いを随所で犯していることが多いものです。

このようなバグを一斉に直すために、バグ原因・修正方法の確認(レビュー)の際に担当者自身で類似バグのチェックを行ったか、どのような方法で行ったか、疑わしい個所が幾つ見つかってそのうち幾つが修正対象になったか、漏れはないか、といった項目を確認項目に加える方法があります。エディタの機能を使ってソースコードを広範囲に検索し、類似パターンを見つけるだけでも効果があります。

第４章

バグ管理内容と管理項目

本章では、バグ管理のために標準的であると考えられる項目の一覧を提示します。項目の選定にあたってはCMU/SEIやIEEE 1044などのグローバル標準を参考に、本書を検討したメンバの組織で使用されている項目も加えて、整合性がとれるよう整理しました。

バグ情報の入力者が何のためにその項目を記述するのかを理解できるように、項目ごとに目的の説明を示しています。

4.1 バグ管理項目の一覧

本節では、バグ管理を行うためにバグに関するどのような情報を用いればよいかを、標準的なバグ管理項目として表 4.1 及び表 4.2 に示します。

表 4.1、表 4.2 の項目については、以下の通りです。

「項目名」　：バグ管理のための情報項目の名称を示しています。グローバルな開発体制を考慮して、英語名も併記しました。

「説明・属性」　：この管理項目が、どのような情報であるのかを示しています。
また、その項目に対する属性情報として、どのようなものがあるのか具体的な例を示しています。

「本項目の目的」：この管理項目の利用目的と利用効果、必要性をプロジェクトの中のバグ管理業務（判断や対策）の視点と、組織としての品質の見える化や改善業務の視点から具体的に解説しています。

ここで示した管理項目を必ずしもすべて採用する必要はありません。別の項目を追加することも可能です。解析修正担当者、テスト担当者、バグ管理担当者、品質管理担当者などのバグ管理に利用可能なリソースの量と、組織としての品質能力改善の必要度合いなどを勘案して、その組織のすべてのプロジェクトで採用する管理項目と個別のプロジェクトで追加採用する管理項目を決めるのがよいでしょう。

表 4.1　管理項目一覧

項目名		説明・属性	本項目の目的
日本語	英語		
管理番号	ID	管理のための番号。	バグをユニークに識別、管理するための情報。
概要 (タイトル)	title	概要を1行程度で示す。	・バグの内容を大まかにつかむための情報。 ・タイトルの付け方をルール化すると検索しやすくなる。
プロジェクト名	project name	対象プロジェクトの名前を記述する。	・プロジェクトごとにバグを1種として取りまとめるための情報。 ・バグ情報をプロジェクト単位で管理する場合は無くてもよい。
重要度 ※製品／顧客視点	severity	重要度を、製品または顧客の視点で示した分類。 (例)S：最重要、A：重要、B：、C：重障害、中障害、軽障害	・バグが与える影響の度合いを分類で示すことにより、分類別にバグを管理して、絞り込むための情報。 ・品質状況把握、修正の優先順位付け、出荷判定に利用するなどの顧客視点での判断を行うために用いる情報。
優先度 ※プロジェクト視点	priority	プロジェクト管理視点でのバグ修正の優先度を示した分類。 (例)高、中、低	・プロジェクト管理視点でのバグ修正の優先順位を明確にするための情報。 ・製品または顧客としては重要でない機能に関するバグでも、そのバグが評価(テスト)など開発工程を止めてしまうような場合、最優先とするなどして用いる。
ステータス	status	対応の状況を記述する。 (例) 起票済(new)、担当者割当済(assigned)、調査済(analyzed)、処置済(resolved)、検証済(verified)、完了(closed) ※上記ステータスの意味などについては、「第3章 バグ管理プロセス」を参照。 ※“再現待ち”が多くあるような場合は、“再現待ち”というステータスを設けることもある。	当該バグについて、対応開始から完了までの状態(状況)を把握、管理するための情報。

項目名		説明・属性	本項目の目的
日本語	英語		
同一原因管理番号	duplicate ID	同一原因の管理番号を記入する。同一原因のバグがない場合には、空欄とする。 ※利用方法など詳細は、「第5章 5.2 バグ１件の数え方の指針」を参照。	本票バグと関連するバグの有無や、関連するバグＩＤ（重複バグ）を把握するための情報。
担当者、日付			
発行者	opened by	本票の作成者。	バグ票（本票）を登録記載した人を明確にするための情報。
発行日	date opened	本票の発行日。 必要であれば、時間まで記録する。	・本票の発行日を明確にするための情報。 ・バグ発行日から完了日までの日数から対応期間や滞留期間などを算出し、バグ対応時間実績を把握する。これにより、対応状況や今後の見通し（判断）、次プロジェクトの計画立案への情報などとして活用することが可能となる。
発見者	observed by	バグの発見者。 発行者と同じ場合、同じ名前を記入するか、省略するかなど、プロジェクトにより運用が異なる。	・バグの発見者を明確にするための情報。 ・バグ原因調査（再現）の際、内容欄に書かれた情報だけでは不明、あるいは再現できない場合、プロジェクトリーダなどが問い合わせを行うための情報として用いることもある。
発見日時	date/ time observed	バグの発見日時。 発生日時が重要な場合、別項目として発生日時を追加する場合もある。	・バグの発見日時を明確にするための情報。 ・バグ発見日から処置完了日までの日数から対応期間や滞留期間などを算出し、バグ対応時間実績を把握することができる。例えば、これにより、対応状況や今後の見通し（判断）、次プロジェクトの計画立案への情報などとして活用することが可能となる。

項目名		説明・属性	本項目の目的
日本語	英語		
調査担当者	analyzed by	調査の担当者。 大規模でない場合やバグトラッキングシステムを使用する場合は、単に「担当者」(assigned to)として示されることも多い。	調査担当を明確にするための情報。
調査日	date analyzed	調査を終了した日付。	バグ調査を終了した日を明確にするための情報。
処置日	date determined	処置方針を決定した日付。	処置方針を決定した日を明確にするための情報。
処置担当者	resolved by	処置の担当者。	・処置担当者を明確にするための情報。 ・処置担当の負荷管理などに使うこともある。
処置期限	due date	処置の対応期限。	・バグの処置の対応期限を明確にするための情報。 ・顧客、他部署など関係者に対して明示(コミット)するための情報として用いる場合もある。
処置日	date resolved	処置が完了した日付。	処置が完了した日を明確にするための情報。
処置承認者	approved by	処置方針、及び処置について承認を行う者。	・処置内容の承認者を明確にするための情報。 ・処置内容に対して、誰が承認したのか(責任を持つのか)を明確にするための情報。
検証担当者	verified by	処置結果の検証担当者。	・処置結果の検証者を明確にするための情報。 ・発見者である場合が比較的多い。
検証日	date verified	処置結果を検証した日付。	処置内容(処置結果)を検証した日を明確にするための情報。
完了日	date closed	処置内容の検証が終了し、処置完了した日付。	正式に処置が完了したことを示すための情報。
リリース日	date released	処置を完了した版をリリースした日付。	バグの処置が完了した版をリリースした日を顧客、他部署などの利用者(開発関係者)に対して明示するための情報。

項目名		説明・属性	本項目の目的
日本語	英語		
内容			
内容	description	・詳細な説明。問題動作だけでなく、本来(仕様として)期待される動作も記述する。 ・問題動作を示すデータがあれば、そのデータへのリンクも示す。データを示す項目は別項目としてもよい。	・バグ発見者がバグ事象を記載するための項目。 ・バグ事象の理解(現状認識)、原因調査、調査のための再現を行うための情報。 ・発生事象の調査、再調査により、追記されることもある。
機能名/サブシステム名	asset: product, component, module	バグが発生した機能ブロックまたはサブシステム。	・バグ事象の理解(現状認識)、原因調査のための情報。 ・調査担当者に対して、発生箇所を具体的に提示することにより、原因調査が行いやすくなる。 ・どの箇所でバグが多く発生しているかなど、バグ分析にも用いる。
発見版数	version detected	・バグを発見した版数。 ・発見工程により対象が異なり、モジュールに対する版を示していたり、製品(ソフトウェア全体)に対する版を示していたりする場合もある。 (例)開発中：モジュールの版、リリース前後：製品の版	当該バグが、どの版(バージョン)で発生した情報なのかを明確にし、調査の際に再現されるための情報。
発見環境	environment	・バグを発見したプラットフォーム。 ・OS、ミドルウェア、コンパイラ、ハードウェアなど、ソフトウェアが動作する環境に関する情報。	当該バグが、どのような環境(組み合わせ)で発生したものかを明確にし、調査の際に再現させるための情報。
発生頻度	frequency	バグの発生頻度を記述する。 (例)不明、常に発生、 1回発生/10回実行　程度、 1回発生/100回実行　程度、 1回発生/100回実行　以下、 その他(n回発生/m回実行または n回発生/m時間実行)	バグ修正の"優先度"を設定するための一つの判断基準として活用する。

項目名		説明・属性	本項目の目的
日本語	英語		
発見工程	detection phase	バグを発見した工程。 (例)実装(コーディング)、 単体テスト(単体テスト)、 ソフトウェア結合テスト (ソフトウェア結合)、 ソフトウェア総合テスト (ソフトウェア適格性確認テスト)、 システム結合テスト (システム結合)、 システムテスト (システム適格性確認テスト)	・バグ全体の分析を行う際に用いる情報。 ・作り込み工程と含めて、より前の工程でバグ検出をするために、改善に注力すべき工程を明確にすることができる。
発見手段	detection activity	バグを発見した(バグが発見された)手段。 (例)コードレビュー(コードチェックツール)、テスト、運用、その他 ※プロジェクトにより、コードチェックツールで検出したものもバグとする場合もある。	どの手段でより多くのバグを見つけているかなど、バグ分析を行うために用いる情報。
発見テスト項目番号	test item ID	テストで発見した場合、発見したテスト項目番号を記載。	・バグ原因調査、再現のための情報。 ・テスト項目番号を容易に知り得ることで、発生手順や関連する情報の取得などが容易になる。 ・また、テスト項目(計画、テスト分類)とバグとの関連から、当該バグがテスト計画に影響を及ぼすかを判断するための情報となる。
影響	effect	どのような影響があるのかを記載。 (内容例)機能性/使用性/セキュリティ/性能/その他	本票バグが品質にどのような影響を与えるかを示す情報。品質特性。

項目名		説明・属性	本項目の目的
日本語	英語		
調査結果			
発生原因	cause	バグ発生の原因分析結果。(調査担当者や、処置担当者が分析した発生原因)	・バグ発生の原因を記録するための情報。横展開にも利用される。 ・調査担当者と処置担当者とが異なることが多いため、調査担当と処置担当の原因分析結果を別々に記録したい場合、別項目として定義するとよい。個別に記録が必要でない場合は、1項目として定義、記録するとよい。
原因箇所	defect found in	・バグが発生した原因を含むソフトウェア成果物。 ・仕様書であれば仕様書名(ファイル名)とその頁、行数など。ソースコードであれば、ファイル名、関数名、行数など。	・バグ発生の原因箇所を明示するための情報。 ・バグ分析にも用いる。
見積もり工数(人H)	estimated effort (person-hour)	修正する場合の工数の見積もり。 ※単位は、一般的には、"人×時間"を用いることが多い。	・修正に必要な見積り工数を明確にしておくことにより、"重要度"、"優先順"と合わせて対応する優先順を検討し、判断する際に用いる情報。 ・リリース期限までの時間(保有するリソース:工数)の中で、対応優先順を考慮する際に、リソースと対比するために用いる。
解決内容			
解決方法/処置内容	resolution	・解決方法、修正内容あるいは対応方針。 ・修正方針もここに記述し、修正内容を追記する。	・バグの発生原因箇所の修正方法、利用の仕方の変更など、このバグを解決するための方法、方針について記載するための項目。 ・バグの発生原因箇所を修正せず、他の箇所を修正したり、運用方法などソフトウェア修正を行わずに解決する場合は、特に"解決方針"として本欄に記載する。

項目名		説明・属性	本項目の目的
日本語	英語		
処置区分	disposition type	処置の分類。 （内容例）修正／次版以降で対応／処置不要／処置しない	処置残や処置状況を明確にし、バグ対応計画を立てる（状況を判断する）ための情報。
修正対象	changes made to	修正した仕様書名やソフトウェアコードのファイル名。	修正箇所（ファイル）を明示するための情報。
処置済版数	version corrected	バグが解決した製品やモジュールの版数。 ・構成管理システムなど、本票やバグ管理システムとは別に管理している場合もある。	どの版（バージョン）からバグが解決されているかを明確にする（知らしめる）ための情報。

分析のための項目

バグ区分	defect type	バグの分類。 ※例などについては、「第4章 4.2バグ区分」を参照。	・バグの内容を分類した情報。 ・開発プロジェクトや組織内でのバグ全体を対象に、品質評価（バグ傾向分析）を行ったり、改善ポイントを検討する際に用いたりするための情報。
作り込み工程	insertion phase	バグを作り込んだ工程。 （例）システム要求定義 （システム要求分析）、 システムアーキテクチャ設計 （システム方式設計）、 ソフトウェア要求定義 （ソフトウェア要求分析）、 ソフトウェアアーキテクチャ設計（ソフトウェア方式設計）、 ソフトウェア詳細設計 （ソフトウェア詳細設計）、 実装（コーディング）	・バグの本質原因となった設計（書）の誤りや、コード誤りなど、起こしてしまった開発工程を示す情報。 ・バグを生じさせてしまった開発工程を明確化し、対象工程の作業手順や入出力情報（資料）を分析調査し、改善するための情報。

表 4.2　参考管理項目一覧

項目名		説明・属性	本項目の目的
日本語	英語		
調査工数（人H）	investigation effort (person-hour)	バグの原因調査に要した工数。 ※単位は、一般的には、“人×時間”を用いることが多い。	バグ調査に要した実績時間の把握、後発プロジェクトでのバグ調査工数見積り、プロセス改善前後での改善状況把握などのため用いる情報。
処置工数（人H）	disposition effort (person-hour)	バグの処置に要した工数。 ※単位は、一般的には、“人×時間”を用いることが多い。	バグ処置に要した実績時間の把握、後発プロジェクトでのバグ対応工数見積り、プロセス改善前後での改善状況把握などのため用いる情報。
発見すべき工程	detection phase (to be)	本来、バグを発見すべき工程。 （例）単体テスト（単体テスト）、 ソフトウェア結合テスト（ソフトウェア結合）、 ソフトウェア総合テスト（ソフトウェア適格性確認テスト）、 システム結合テスト（システム結合）、 システムテスト（システム適格性確認テスト）	・バグ全体の分析を行う際に用いる情報。作り込み工程と含めて、本来、発見すべき工程を明確にしておくことで、改善に注力すべき工程を明確にすることができる。 ・個々のバグ管理プロセスの中（バグ収束に向けた活動をしている最中）ではなく、プロジェクト完了後などに改善のための分析を行う際に付記（記録）しておき、次プロジェクトに向けての改善、あるいは、当該組織プロセスの改善の際に、注力すべき点を明確化（分析記録）しておくための情報。
発見すべきアクティビテイ	detection activity (to be)	本来、バグを発見すべきアクティビティ。 ※アクティビティ：工程作業を、さらに分割し、順序付けした作業要素。	“発見すべき工程”　と同じ。

●重要度と優先度

重要度は、製品または顧客の視点で分類、レベル付けをしたもので、優先度は、プロジェクト開発管理、開発チームの視点で、分類、優先順位付けをしたものです。

ソフトウェアがフリーズした場合などのように、重要度が大で優先度も大という場合が多いのですが、例えば、固有名詞の誤りなどは、プロジェクト管理（開発チーム）の視点としては他に与える影響が軽微なバグであるため優先度は低くなりますが、顧客視点では重要な事項となります。この場合は、顧客リリースまでしばらく期間がある場合は修正作業はいったん保留としておき、他のバグの修正状況により修正にとりかかることにします。しかしながら、顧客リリース直前または直後だった場合は、最優先で修正作業に取りかかるということがあり得ます。

また、例えば、処理のログ出力機能など顧客の視点では重要ではない機能のバグであった場合でも、そのバグが他の機能の実行（試験）を妨げるような状況であった場合、優先度は高いものとなります。

●担当者と日付

バグ管理対象のプロジェクトが大規模の場合、あるいは、バグ票（Excel など）でバグ管理を行う場合は、表 4.1 で示したようにそれぞれ別項目として設けることがありますが、大規模ではない場合やバグ管理ツールを用いる場合などは、“担当者”という項目だけで管理する場合もあります。

4.2 バグ区分

本項では、バグ管理項目の中の“バグ区分”について説明します。

バグ区分はバグの内容を分類した情報項目で、開発プロジェクトや組織内でのバグ全体を対象に、品質評価（バグの傾向分析）を行ったり改善ポイントを検討したりする際にこの情報を用いることができます。

バグ発生工程別に分類したバグ区分を表 4.3、バグ管理プロセス運用上必要な区分を表 4.4 に示します。

表 4.3、表 4.4 の項目については以下の通りです。

「工程」：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。
「種別」：バグの種類を示しています。
「説明」：バグの「種別」について説明しています。
「例」　：バグの「種別」について例を示しています。

一つのバグについてそのバグ区分を検討した結果、該当する種別が複数考えられる場合は、最も重要と考える種別を選択します。また、いったんは「その他」の種別に分類した後、バグ分析の過程で適切な種別に分類し直すこともあります。

バグ区分は、バグ分析を行う開発プロジェクトや組織の特性を考慮してそれぞれで適切なバグ分析を行うために、必要に応じて選択または変更します。

表 4.3　バグ発生工程別に分類したバグ区分

工程	種別	説明	例
要求定義（外部）	記述誤り（外部）	外部の要求仕様書における記述の間違い、不明瞭、漏れなどによるもの。 ※外部＝顧客、システム、ハードウェアなど	・異常時の動作仕様に間違いがあり、システムとして安全状態に移行しないケースがあった。 ・制御に使用する外部入力信号の定義がもともと 1 本漏れていた。 ・マイコンのポートに割り付けられた入（出）力信号の情報がハードウェア要求で間違って指示されていた。
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の間違い、不明瞭、漏れなどによるもの。	・状態遷移表（図）の遷移条件が間違っていた、または抜けていた。 ・故障検知のためのフェイルセーフ仕様を間違っていた。 ・フローチャートの矢印表記が無かった。
	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。	要求定義（外部）には定義されていた機能が、ソフトウェア要求仕様書でその機能自体の記述が無かった。
	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。	・外部の要求仕様で未定義の内容を勝手な解釈でソフトウェア要求仕様に盛り込み、結果、顧客から間違った仕様であると指摘された。 ・既存のソフトウェア要求仕様書を流用したが、開発中の製品では不要な機能の記述が含まれていた。 ・開発途中の仕様変更で削除された機能がソフトウェア要求仕様書から削除されていなかった。
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。	・取り得るパラメータの範囲が間違っていた。 ・条件判定で使用するデータの種類が不足していた。 ・条件判定で使用するデータの種類が間違っていた。 ・データの初期値が間違っていた。 ・使用するデータ変数名が間違っていた。

工程	種別	説明	例
設計	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。	・データの計算式が間違っていた。 ・計算の順番が間違っていた。 ・意図しない演算のオーバーフローが発生していた。
	インターフェースの誤り	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。	・引数の値が間違っていた。 ・引数の型が間違っていた。 ・戻り値の使い方が間違っていた。
	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。	・データの初期化・更新タイミングが間違っていた。 ・多重割り込み処理において、割り込み処理の優先順位が間違っていた。 ・割り込み許可（禁止）タイミングが間違っていた。 ・関数の呼び出しタイミングが間違っていた。
	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。	・マイコンリソース（例えばレジスタ）の使用方法が間違っていた。 ・mallocのサイズが少なかった。 ・ファイルのフリーをしていなかった。 ・二重解放していた。 ・決められた処理時間内に処理が終わらなかった。 ・ROM、RAMが不足していた。
	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。	・偶数パリティとするところを奇数パリティにしていた。 ・NULLを引数にとる可能性のある関数の設計仕様で、NULLチェックを記述していなかった。
	記述誤り	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。	・フローチャートの矢印表記が無かった。 ・設計書バージョンの記述が間違っていた。
	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。	既存の設計書を流用したが、開発中の製品に必要な機能が抜けていた。
	機能の設計誤り	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。	・既存の設計書を流用したが、開発中の製品では不要な機能の記述が含まれていた。 ・開発途中の仕様変更で削除された機能が、設計書から削除されていなかった。

工程	種別	説明	例
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。	・データの初期化をしていなかった。 ・データの初期値が間違っていた。 ・使用するデータ変数名が間違っていた。
	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。	以下のような間違いをしていた。 ・判定条件として、 (誤)「if (x != 10) {…」 ⇒(正)「if (x == 10) {…」 (誤)「if (x + y) {…」 ⇒(正)「if (x - y) {…」 ・ループ条件として、 (誤)「while (x <= 5)」 ⇒(正)「while(x < 5)」 ・switch文におけるdefault内の処理が無かった、または処理が間違っていた。 ・ポインタ使用時の参照先演算が間違っていた。 ・代入符号「=」と判定符号「==」が間違っていた。
	インターフェースの誤り	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。	・NULLを引数にとらない仕様の関数にNULLを渡していた。 ・呼び出す関数が間違っていた(関数func1を呼び出すところを、func11を呼び出していた)。
	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。	・データの初期化や更新場所が間違っていた。 ・割り込み許可(禁止)場所が間違っていた。 ・関数の呼び出し場所が間違っていた。
	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。	・mallocのサイズ数値が間違っていた。 ・二重解放をしていた。 ・あるリソースの確保には特定レジスタの値を設定する必要があったが、設定が間違っていた。
	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。	・異常判定処理が無かった。 ・CRC生成多項式の値が間違っていた。

工程	種別	説明	例
実装	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。	・設計書に記載されている機能がコード上で存在しなかった。 ・既存のライブラリを流用したが、開発中の製品に必要なコードがライブラリ内に存在しなかった。
	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。	・既存のライブラリを流用したが、ライブラリ内に今回の設計に不要なコードが存在していた。 ・開発途中の仕様変更で削除された機能のコードが削除されていなかった。
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。	・統合すべきモジュールを選択しなかった。 ・統合不要なモジュールを選択した。
	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。	テスト実施時に、 ・参照すべきデータベースの版より古い版のデータベースを利用していた。 ・該当機種と異なる機種のデータベースを利用していた。
	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「インターフェースの誤り」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 4.4　バグ管理プロセス運用上必要な区分

種別	説明	例
修正漏れ＊	過去に発生したバグに対して修正したが、修正内容や修正範囲に漏れがあったため、バグが再発したもの。	バグ分析の結果、修正が必要な3カ所を特定し修正を行った。その後、テストを進めるうちに再び同じバグ事象が発生。バグ分析の結果、もう1カ所で修正が必要な部分が見つかった。
修正誤り＊	過去に発生したバグに対して修正したが、修正内容や修正範囲に誤りがあったため、バグが再発したもの。	バグ分析の結果、修正が必要な3カ所を特定し修正を行った。その後、テストを進めるうちに再び同じバグ事象が発生。バグ分析の結果、修正箇所3カ所のうち1カ所で修正方法の間違いが見つかった。
重複バグ＊	複数のバグ事象の原因が同一の原因と判断できた時点で、これらを単一の現象として扱う。このとき代表して調査を進める事象（親バグ）以外の事象を「重複バグ」（子バグ）として明示化する。	当初3件のバグ事象を発見し、バグ票を起票していた。バグ分析を進めると、バグ発生原因が同じであることが見つかった。
非バグ	最初バグと判定されたものの、バグ分析の結果、バグではなかったもの。	・テスト仕様書通りの入力条件ではない状態でテストを実行したため、テスト仕様書に記載された期待値が得られなかった。 ・テストに使用したハードウェア・バージョンが異なっていた。 ・テスト仕様書に記載された期待値が間違っており、テスト実施結果の方が正しいものであった。 ・テスト結果が本来OKなのに、間違ってテスト結果書にNGと記述してしまった。
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。	―

＊：バグ数カウントの観点から必要な分類。詳細は、「第 5 章 5.2　バグ 1 件の数え方の指針」を参照。

＊参考文献

SEC BOOKS「【改訂版】組込みソフトウェア向け開発プロセスガイド」IPA/SEC、2007 年
"What Types of Defects Are Really Discovered in Code Reviews?", Mika V.Mantyla and Casper Lassenius, 2008

Column

エンタプライズ系からのバグ管理のヒント
～大規模化が進む組込みソフトウェアへの対応～

組込みシステムは、家庭用、産業用、医療用など電子制御を行うほとんどの製品で使用されています。その中でも、多種多様な機能が必要とされるデジタル家電、自動車、携帯電話の分野では、ソフトウェア開発を数百人単位の人数、数年規模の期間で行うケースも珍しくありません。このようなソフトウェア開発におけるテストやバグの対応では、エンタプライズ系のバグ管理手法が参考になる場合があります。

ここでは、エンタプライズ系で特徴のある連携テスト工程 / 運用テスト工程のバグ管理の一例を紹介します。

●連携テスト工程／運用テスト工程におけるバグ管理例

機能単位～機能間連携～サブシステム間連携のテストでは、専任のテスト担当チームを組織してテストを実施します。

また、検出バグの修正においては、お客様も参画する修正審議グループによる審議を行います。システムが大規模であり、バグ修正がシステム全体の品質や進捗にどのように影響するか、見極める必要があるためです。

○体制

・テスト担当チーム
・テスト管理者
・修正審議グループ
　※開発責任者、運用責任者、お客様がメンバとなる
・構成管理チーム
・テスト支援チーム（バグ票管理ツールの運用チーム）

○バグ対応手順

システム規模が大きいため、検出バグの修正は以下①～⑥の手順で実施します。

①バグ票の起票　［テスト担当チーム／開発担当者］

バグ票を起票し、バグ修正案・影響範囲を記載する

②バグ票の確認 ［テスト管理者］

バグ票のバグ修正案・影響範囲をチェックし、審議提出可否を判断する

③バグ修正案の審議 ［修正審議グループ］

影響範囲の正当性の確認、修正の妥当性の判断、リリース時期の調整を行う

④バグ修正 ［テスト担当チーム／開発担当者］

上記③で確定した修正案に従いソースコードを修正し、確認テストを実施する

⑤クローズ確認 ［テスト管理者］

修正確認エビデンスをチェックし、バグ票をクローズする

⑥リリース ［構成管理チーム］

修正プログラムのリリース（システム再ビルド）を行う

○主なバグ票記載項目と補足資料

バグ票に記載する項目については「第４章 バグ管理内容と管理項目」を参照してください。さらに、上記「バグ対応手順 ③バグ修正案の審議」のため以下の資料を作成して、バグ票から参照できるようにします。

・テスト仕様書（テストケース、テストデータ、操作手順説明、テスト結果期待値）
・テスト結果（不具合を示すメッセージ、ログ、DB データ、画面）
・バグ修正説明書（プログラム･設計書の修正案詳細、影響範囲説明）
・修正確認テスト説明書（修正結果・影響範囲の正当性確認の方法を説明）

○ツールによるバグ対応手順の管理

バグ票の起票～修正審議～修正・確認テスト～リリースのプロセスに係わる人数が多いため、資産管理ツールを用いて、上記「バグ対応手順①～⑥」の各作業で作成・更新する資料にアクセスできるメンバを限定しておきます。バグに対応する際の作業（手順）の誤りを防止し、ルール通りのプロセスを確実に実施できるようになります。

この方法は、設計工程や製造工程での成果物作成にも利用でき、資産の保全、及び進捗管理の効率化を支援します。

第5章

バグカウントの指針

本章では、バグの件数をカウントする指針として、レビューやテストで指摘される問題のうち、何をバグであるとしてカウントの対象とするのか、またバグ1件とはどのような考え方でカウントするのかについて説明します。

バグのカウント方法の統一は、組織単位でもプロジェクト単位でも効果を得られますが、要求仕様に対する条件を絞り込めるプロジェクト単位で行う方が効果は大きいでしょう。

ソフトウェアに内在するバグの数を把握することは、そのソフトウェアの品質や開発チームの能力成熟度の評価に役立ちます。しかし、そのためには、何をバグとするか、また、1件のバグはどのようにカウントするのかについての考え方が開発組織やプロジェクトチーム内で揃っている必要があります。この考え方がばらばらだと、見かけのバグのカウント値に影響されて正確な評価が困難になります。極端なケースでは、開発工程の次のステップに進む条件として「発見すべきバグ数のガイドライン」が設定されている場合に、恣意的にバグを増やしたり、1件のバグを複数件に数え直したりすることさえ起こりかねません。このようなことが起こらないように、プロジェクトの開始前に統一したバグのカウント方法を決めておくことが重要です。

5.1 バグとする問題の指針

本節では、どのような問題をバグとするかについて指針を示します。

●原則

何をバグとするかについて、本書で推奨する原則は次の通りです。

仕様と異なる動作を引き起こすプログラムの誤りをバグとする。

しかし、発見した問題のうち何をバグであるとするかは、ソフトウェアの品質確保（のレベル）やプロセス改善、開発チームのスキル向上など、バグカウントの目的に依存するところがあります。バグであるとするか否かを目的に依存して定める例を、次項で簡単に説明します。

ソフトウェア成果物の品質評価への信頼性をより向上させるため、品質評価資料にこのようなバグカウントの指針についても含めることを推奨します。

●目的に応じてバグとする問題を考えるための指針

（1）コーディング規約など、プロジェクト標準への違反をバグとするか？

例えば保守性が重要なプロジェクトの場合、コーディング規約などの標準への違反をバグとして扱うこともあります。その場合、これらをバグとすることで開発チームのメンバに問題の重要性を意識させ、きちんと修正することを狙いとしています。また、開発チームのスキル向上に重きを置いている場合にも、プロジェクト標準への違反をバグとして扱い、重要性を意識させ、また統計的

に処理することで組織としてのスキルの状況を把握する場合もあります。

ただし、このようなプロジェクトでも、動作に影響の無い問題をバグとして修正することは単体テスト工程までに止めるべきです。結合テスト以降では、動作に影響の無い問題はデグレードする危険性を考えると、修正するメリットはほとんど無いと言っても過言では無いでしょう。

（2）コメントの誤りをバグとするか？

コメントの誤りはその後のプログラムの保守に大きな影響を与えますので、単体テスト工程までに発見されたものであれば修正することが望ましいでしょう。その場合、標準への違反と同様にバグとして管理することも考えられます。

（3）冗長なコードをバグとするか？

冗長なコードはコメントの誤りと同様に、その後のプログラムの保守に影響を与えますので、単体テスト工程までに発見されたものであれば修正することが望ましいでしょう。その場合、標準への違反と同様にバグとして管理することも考えられます。

（参考）対象とする成果物について

バグカウントの対象となるソフトウェア成果物は、一般には、リリース対象のプログラム一式であり、プログラムのテスト手順やテストに利用するプログラムなどは対象とはしません（これらに関する問題は、バグ票では「非バグ」として分類されます）。テストの誤りは、別途カウントしてプロセス改善などに活かすとよいでしょう。

Column

大規模開発における留意点 その3
バグを適切な担当者に振り分ける

発見されたバグの解析・修正の仕事を誰（グループ／個人）に割り当てるかについては、誰がどのように決めているでしょうか。不適切な人に振り分けると、原因究明ができずに放置されるか、よくても、いったん調べた上で差し戻しされ振り出しに戻ることになります。これが一つだけであればまだしも、幾つも同様のことが起こると、期限（納期）内に製品が完成しない、下手をするといつまでも完成しないといった最悪の状況に陥る可能性があります。適切な人に確実に振り分けることが製品の完成にとって大変重要です。

一方、振り分け作業ができる人が、技術レベルの高い人や製品を熟知している人に限られるようでは、大量にバグが発見されて納期も迫っているような状況でプロジェクトが頓挫しかねません。それなりの人がそれなりの時間内に、適切な振り分けができる必要があります。それなりの人とは、技術力や製品の熟知度が高くなくても、極端に言えば低くても、間違った振り分けが許容できる頻度に抑えられる水準であれば OK です。もちろん何の知識も無く何の判断基準も与えられなければ、当然振り分けはできません。

そこで必要になってくるのが、バグ発見時の付加情報収集とその付加情報を使ったバグ振り分け（判断）ルールの整備です。付加情報には、問題が発生した時に利用していた製品の外的機能（ユーザにとっての機能）と動作していた内部機能（データの取り込み / 出力や表示など）などがあります。振り分けルールとは、どの付加情報が何であれば次に何の情報をみていくといった、一連の if-then rule のセットです。

もちろん、判断誤りはある程度の確率で発生しますが、このような振り分けはバグの解析・修正作業を円滑に進めながら限られた時間内で作業を終わらせるためには必要な対策であり、プロジェクトの規模が大きくなればなるほど有効な対策になります。

5.2 バグ１件の数え方の指針

本節では、バグ１件の考え方について指針を示します。

●原則

本書で推奨するバグ１件の数え方の原則は、以下の通りです。

- ●設計仕様書やソースコードなどのソフトウェア成果物に含まれるバグの原因部分について１件とする。
- ●バグ原因がプロセスにある場合は、対応する修正箇所を個別に数える。

●バグカウントで注意すべき点

テスト工程でのバグ現象の発見から修正適用までのプロセスフローは、「第３章 バグ管理プロセス」で説明される通りです。ここで、バグカウントの信頼性を確保するために、プロセスフローの初期段階で注意すべきポイントが幾つかあります。

（1）カウントから漏れてしまうバグを無くす

最初のポイントはカウントから漏れてしまうバグを無くすことです。よく起こるのは、単純なバグで原因も修正方法も明白なために設計担当者の一存で修正を行ってしまい、バグ情報が管理担当者に報告されないケースなどです。開発プロジェクトの運用規約などを定めて、このような漏れを防止する必要があります。

（2）再現しないバグを取り消す

次のポイントは、再現しないバグの扱いです。バグ現象が再現しない場合は「再現待ち」と属性指定して一時保留します。長期間経過しても再現しないときには、期限を区切って「再現せず」などに変更してバグとしての扱いを取り消さないと、いつまでもバグ数に未確定要素が残ることになります。

（3）本当にバグであることを確認する

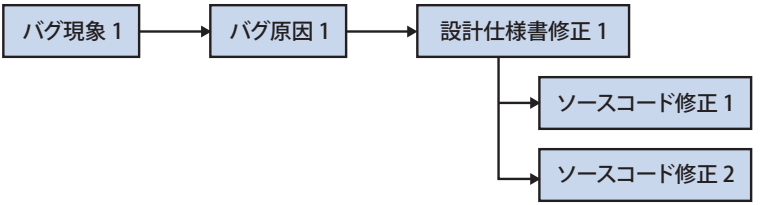
３つ目のポイントは、報告されたバグが本当にバグであることの確認です。テスト工程でバグを発見するのは主にテスト担当者ですが、仕様の理解不足による誤判定やテスト実行環境の設定ミス、テスト用データの不良などにより、

仕様通りの正しい結果であったものがバグとして報告される場合があります。これらも「非バグ」や「仕様通り」などの属性指定を行って、バグとしての扱いを取り消す必要があります。

これらのスクリーニングを通り、バグであることが確定した現象を次の原因調査と修正のプロセスに移します。ここから幾つかの事例に基づいて「バグ1件の数え方」を説明しますが、原則は先に紹介した「成果物に含まれるバグの原因部分について1件とする」ことです。

●事例1：単純な場合

あるソフトウェアの単体テストで発生したバグ現象1件の調査で、設計仕様書の誤りを発見し、当該箇所1カ所を修正した後、これに伴って関連するソースコードの2カ所を修正した。



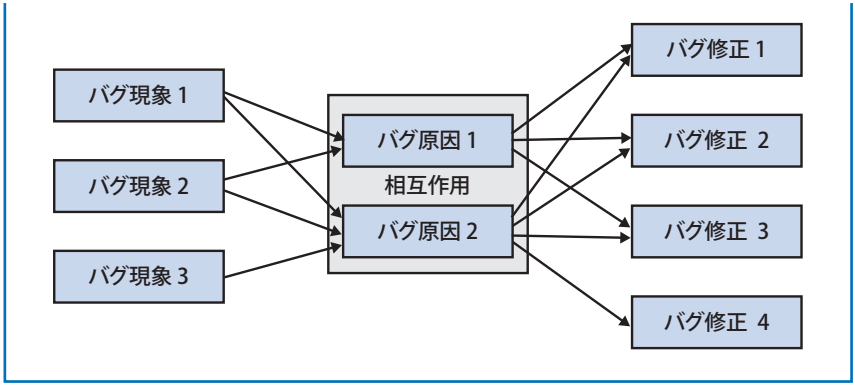
修正箇所の数にかかわらず、1件のバグ原因に対してバグ1件と数えます。バグの原因調査と修正の進捗状況は対応するバグ票に逐次記録して管理されますので、基本的にはバグ票の発行数がバグの数を表し、

バグ数 ＝ バグ票の数　⇒　バグ1件

となります。

●事例2：重複バグの場合

あるソフトウェアシステムの結合テストで発生した3件のバグ現象を調査したところ、2つのサブシステムで設計誤りが見つかり、その相互作用が直接の共通原因となっていることが判明した。このために、該当する設計仕様書とソースコードの4カ所を修正した。



これはテスト工程の後半によく見かける事例です。1 次的には 2 つのサブシステムに設計誤りがありましたが、ここまでバグ現象が発見されておらず（潜在バグ[注]）、ここで発現した 3 件のバグの直接の原因はそれらの相互作用です。ここに着目して、1 件のバグがあったと考えます。

本事例の時間的な経緯を見ると、当初 3 件のバグ現象を発見してバグ票を起票した時点では、バグ数は 3 件とカウントされます。その後、調査が進んで同一原因によるものと判断した時点で、3 現象の一つ（通常は最も調査が進んだバグ現象）を「親バグ（original）」とし、残る 2 つを「重複バグ（duplicate、子バグ）」と属性指定して、バグ数を 1 件とカウントし直し、

バグ数 ＝ バグ票の数 － 重複バグの数 ⇒ バグ 1 件

となります。

（注）不具合が発現していなくてもプログラムの中に潜んでいる場合、「潜在バグ」などと呼ばれます。

相互関連するバグのカウントについての補足

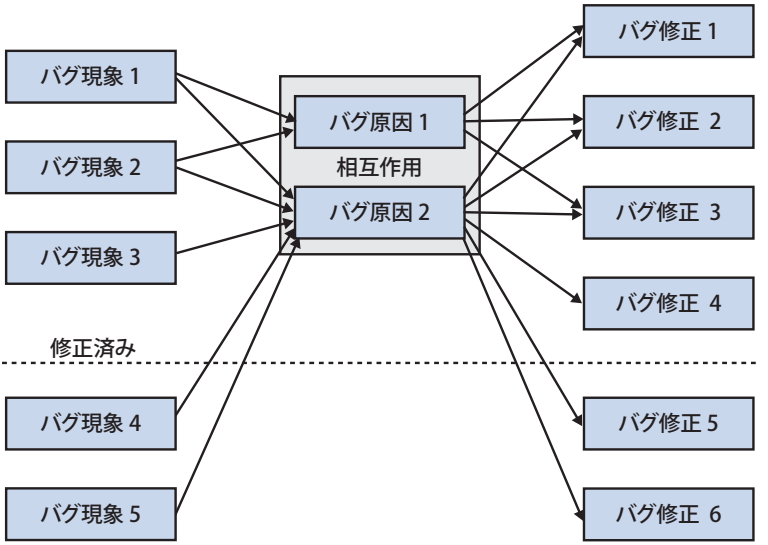
本事例で、元から存在した 2 つのサブシステムの設計誤りはそれぞれが（潜在）バグですが、これらを個別のバグとしてカウントするか否かは、その開発プロジェクトの運用規約で定めればよいでしょう。カウントする場合には新たにバグ票を起票するので作業が増えますが、そのサブシステムがライブラリなどで他のシステムに流用されている場合には、バグの存在を他の開発プロジェクトにも明確化できる利点があります。

重複バグの管理についての補足

重複バグがあるときは、親バグと重複バグ間でお互いを明示的に参照できるようにしておきます。通常、重複バグでの原因調査と修正は中断されて親バグでの修正完了を待ちますので、重複バグのバグ票に親バグの管理番号を記載してその作業進捗を参照できるようにします。一方、親バグの修正が完了した段階で、同一原因と想定された重複バグの不具合が解消していることを確認する指示をそれぞれの担当者に出す必要がありますので、親バグのバグ票にすべての重複バグの管理番号を記載しておきます。

●事例3：修正漏れの場合

事例2のバグの修正完了後に、別の結合テストで同様のバグ2件が発生し、調査の結果、ソースコードに修正が必要な箇所がもう2カ所発見された。それを修正して該当するバグが解消された。



本事例は、事例2の修正を行った時点で「修正漏れ」というプロセス起因の新たなバグが作り込まれたものと解釈します。また「修正漏れ」はそれぞれの箇所で個別に発生したと考えて、別々にバグをカウントします。こうすることにより、同一工程で発見した「修正漏れ」をまとめて1件と数える方式に対して、

「修正漏れ」を発見したテスト工程が同じでも異なっていてもカウント値に違いはなくなり、バグの数え方の一貫性を保つことができます。

5.2 節の最初に挙げた「成果物に含まれるバグの原因部分について 1 件とする」という原則は、開発の上流工程で作り込まれたバグに関係するのに対して、この事例でのバグは下流工程で作り込まれています。ソフトウェアの品質や開発チームの能力成熟度を品質改善の目的で評価するには、工程別などの属性を区別して集計することが役立ちます。このため、「バグ原因がプロセスにある場合は、対応する修正箇所を個別に数える」を原則に加えています。そこで、

バグ数 = バグ票の数　⇒　バグ 2 件　（バグ現象 4 および 5 の 2 件）

となります。

Column

大規模開発における留意点 その4
重複バグの発生を抑制する

プロジェクトの規模が大きくなると、大勢の担当者がテスト作業を分担して行わざるを得なくなります。その時に発生してくるのが重複（duplicate）バグと呼ばれるバグです。これは原因が同じであるバグのことで、複合機能テストや性能テスト、ランダムテスト、耐久テスト、実用テストなどを集中して実施すると必ずといっていいほど発生します。

製品品質の熟成度合いやバグの修正速度、テストの並行度合いによって大きく異なってきますが、プロジェクトによっては最終的に発見されたバグに占める重複バグの割合が数十パーセントに達することもあります。

あるバグが重複バグであるかどうかが一目で判断できれば問題はたいして大きくはなりませんが、時間をかけて調査した結果重複バグであると判明することが大半であり、原因究明にかけた貴重な労力が無駄遣いに終わってしまいます。重複バグの数が少なければプロジェクトの進捗に大きな影響は出ませんが、重複バグがバグ全体の数十パーセントも占めるとなると、納期・開発費に与える影響は致命的にもなりかねません。このような状況を引き起こさないようにするためには、以下の点に留意することが肝要です。

（1）複合機能テスト（製品全体のテスト）は、各機能のテストの結果、品質が熟成したと判断できるまで開始しない。

（2）ランダムテストや耐久テスト、実用テストなどは、並行度合いをむやみに上げて行わない。

数十から100台にも及ぶような同時並行テストはやらない（並行度合いを上げると早くバグを見つけることができると思われがちであるが、実は逆効果である）。

第６章

バグの分析

本章では、バグの分析について、第4章で挙げた管理項目を用いて、何について、どのようにして評価するのかについて説明します。

6.1 分析の目的

バグの分析は、大きくは以下の二つの目的で行われます。

（1）バグの検出状況によるソフトウェア品質の推定

●現工程作業の制御

バグの分析をすることで、あとどのくらいのテストを実施すれば現工程を終了できる品質に到達できるかを予測します。その際に、サブシステムやチームごとに見て品質に偏りがある場合、改善のためにテスト計画の見直しや前工程の再実施などの品質向上策を実施します。

●後工程・最終品質の予測

バグの分析結果から、後工程や稼働時の最終品質を予測します。それにより後工程の作業内容や作業スケジュールの調整を行い、最終品質が要求レベルに達するようにします。

（2）バグの分析によるソフトウェア開発のカイゼン

●振り返りによる問題の特定と対策

バグの作り込み要因や前工程で発見できなかった要因を分析し、ソフトウェア開発プロセスの問題を特定し、カイゼンに繋げます。

6.2 分析手順

バグの分析を行うためには、分析対象のメトリクス（評価尺度）が必要になります。多くのメトリクスを収集し、様々な手法を活用した分析ができることが理想です。

ただし、多くのメトリクスを収集するということは、現場に負担がかかりコストが増加します。そのため、はじめから多くのメトリクスを集めることよりも、まずはできる範囲で少ないメトリクスを活用した分析を実施し、徐々に収集するメトリクスを増やしていくことをお勧めします。メトリクスを集めることが目的ではありません。

本節では、分析対象のメトリクスを少しずつ増やした場合の分析方法を、幾つかの事例を交えながら説明します。分析に関しては、「定量的品質予測のススメ」（IPA/SEC、株式会社オーム社、2008 年）や「【改訂版】組み込みソフトウェ

ア開発向け品質作り込みガイド」（IPA/SEC、2012 年）にも記載がありますので参考にしてください。

分析手順としては、バグのメトリクスだけでの実施から、他のメトリクス（テストケース数やソースコード行数など）と組み合わせた場合の分析方法を図 6.1 及び図 6.2 で説明します。

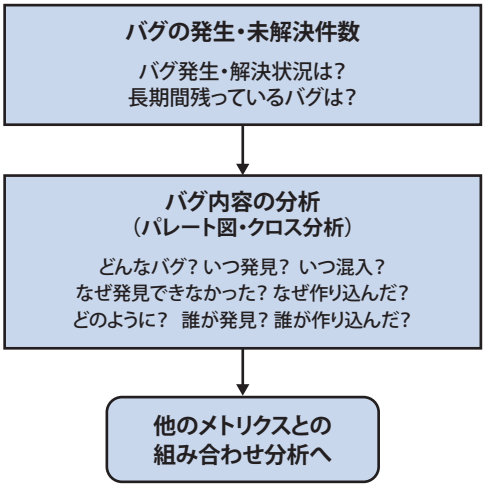


図 6.1　バグの分析手順

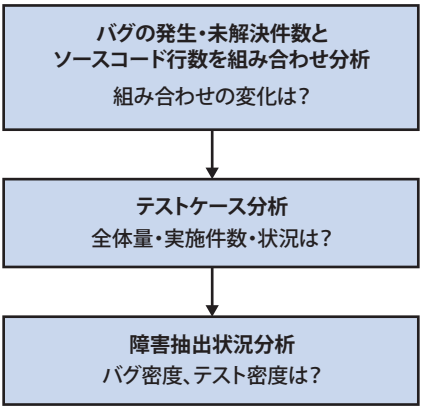


図 6.2　メトリクスの組み合わせによる分析

分析する際には、以下の点を考慮することでより良い分析結果が得られます。

- **工程の開始日・終了日やマイルストーンを考慮した分析**
- **全体だけでなく、サブシステム単位などの詳細な分析**
- **ソースコードの行数は、流用、改造、新規、自動生成、テスト用コードなどを区別**

6.3 分析方法

続いて、分析方法について事例を交えながら説明します。

テスト工程のバグ分析では、要件、仕様から実装に至る様々な上流の工程で混入され、結果として実装までで除去できなかったバグを、テスト工程でどのくらい除去できたか、またあとどのくらい残っているかを予測します。

テストが単体テスト、結合テスト、総合テストと進むほど、発見されたバグに対して、その原因の特定から対策の検証まで膨大な手戻り工数が必要になります。従って、テスト工程全体を通して「実施している工程で発見可能なバグはできるだけ同じ工程で発見し、除去する」ということが重要になります。

テスト工程を制御するためには、工程の最後になって初めて品質状況を評価

するのではなく、テストの進捗と同期して評価する必要があります。

プロジェクトの状況は刻々と変化し、プロジェクトによって収集される様々なデータに直接的あるいは間接的に反映されていきます。その中から、特に悪い方向に向かう変化をいかに早い段階でとらえるかが重要になります。

(1) バグ発生件数・未解決件数

バグの発生件数と未解決件数（バグの残件数）を測定した場合の事例を紹介します。ここでは表 4.1 に示した管理項目のうち、以下の項目を利用して、バグの発生状況や対応状況からテスト工程の品質を分析します。

●利用するバグ管理項目

No	バグ管理項目	利用目的
1	管理番号	バグの件数を計測するため
2	発行日／発見日時	バグがいつ発生したかを計測するため(発生件数)
3	処置日／完了日	バグがいつ解決したかを計測するため(未解決件数)
4	機能名／サブシステム名	サブシステムごとに計測するため

●システム全体の分析

バグの発生件数・未解決件数の推移をシステム全体で分析します。

全体の推移状況から、変化点（急激な変化や停滞を起こしている点）を見つけ出します。変化点を見つける際には、工程・マイルストーンを考慮することで見つけやすくなります。例えば、「テスト工程の終了間際で多くのバグが発生している」や「テスト工程の終了間際にもかかわらず未解決のバグが残っている」などが工程と変化点との関連になります。

以下、事例をもとに説明します。

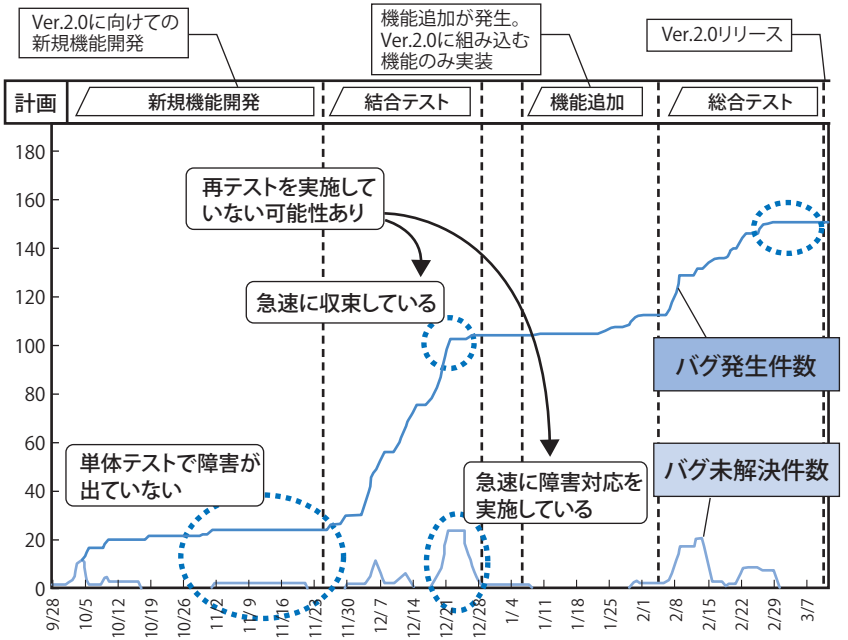


図 6.3　バグ発生件数・未解決件数（全体）

バグの発生件数・未解決件数の推移に対して、計画の工程を合わせてグラフ表示しています。図 6.3 で、点線で囲った箇所の変化点に着目した分析を紹介します。

- 新規機能開発（実装工程）の後半を見ると、バグの発生件数が停滞しています。単体テストが実施されていないことが予想されます。次工程以降において、本来単体テストで発見すべきバグが発見された場合、膨大な手戻り工数が発生します。
- 結合テストの終了間際を見ると、バグの発生件数が突然収束しています。また、バグの未解決件数も急激に減少しているのが分かります。テスト期間に合わせてテストを終了しようとしている可能性があると考えられます。もしそうであれば、解決したバグに対する再テストを実施していない可能性があり、品質に不安が残ることになります。
- 総合テストの終了間際を見ると、バグの発生件数が落ち着いてきていることが分かります。未解決件数も 0 になっていることから、リリースに向けての準備が整っていると考えられます。バグの収束に関しては、84 頁のコラム「バグの収束とバグ曲線」を参照してください。

●サブシステム単位の分析

バグの発生件数・未解決件数の推移をサブシステム単位で分析することで、全体からは分からなかった詳細な分析が可能になります。サブシステム単位で分析する際には、サブシステムごとの複雑度や難易度を考慮するとより良い分析になります。

以下、システム全体の分析で使用した事例（図 6.3）について、このシステムを主要なサブシステム単位に分割した事例をもとに説明します。

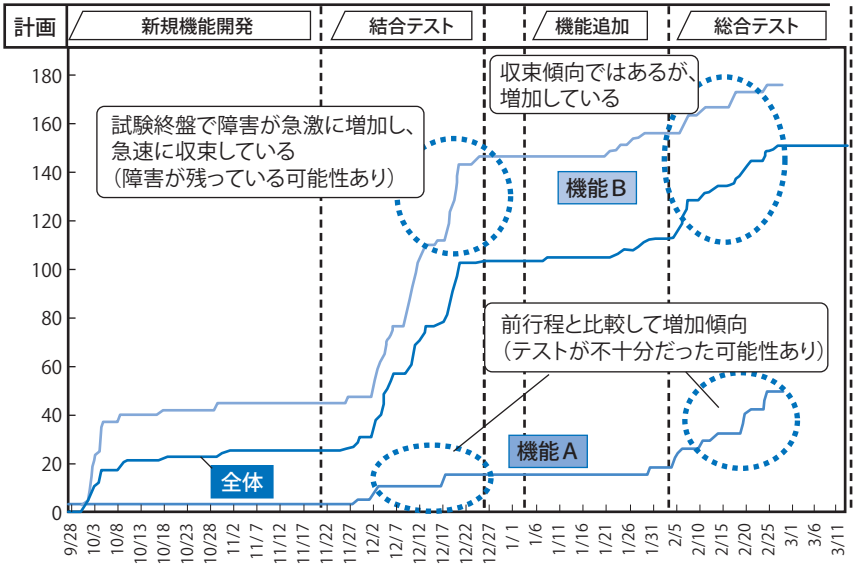


図 6.4　バグ発生件数・未解決件数（サブシステム別）

図 6.4 で、点線で囲った箇所の変化点に着目した分析を紹介します。

- 結合テストの終盤でバグが急激に増加しているサブシステムは機能 B であり、機能 A ではあまり発生していないことが分かります。これに対して二つの観点で分析ができます。一つは、単純に機能 B の方が多くバグが出ているため品質に問題がある可能性が高いということです。もう一つは、機能 A のテストケースが不十分でバグが見つけられていない可能性があるということです。
- 総合テストと結合テストを比較すると、機能 A は結合テストより総合テストでバグが多く発生していることが分かり、テストが不十分であった可能性が高いと考えられます。

Column

大規模開発における留意点 その5
ブロッキングバグは最優先に集中して駆除する

数万から数十万件に及ぶテスト項目を効率よくこなすためには、リセットやフリーズといった、製品の動作を停止させてしまいテストの続行を困難にするバグをいち早く発見して解決する必要があります。そのための方法として、以下のようなものが考えられています。

（1）早期に発見するためのテスト戦略

①外部機能（利用者から見た機能）から

システムが提供する機能を分類・整理し、これまでの経験に基づいて重大バグが発生しやすいと判明している機能から先にテストする。

②内部機能（実装するための機能）から

ソフトウェアが責務とする機能の中で、リソースを扱う（ロックや競合が発生しやすい）部分から先にテストする。

③変化点の多いソフトウェアから

差分開発で変化点の多いソフトウェアから先にテストする。ただし、テストの実施可能性を十分に留意すること（自然現象はコントロールできない。模擬環境でどこまで網羅できるかの検討を含める）。

（2）最速で解決するための方法

①バグのプロファイリング

バグに重要度（システムに与える影響の度合いでランク付けする）と優先度（処置の優先度をランク付けする）を付けて管理する。

②スワットチームによる支援

・テストと並行して、（1）でテストの優先度を高くしたソフトウェアをハイレベルのエンジニアで構成するスワットチームが集中的にレビューする。

・重要度の高いバグの解決に際しては、発見された原因と対策案に対するスワットチームによるレビューを必須とする。

（2）バグ内容の分析

バグの発生件数・未解決件数の推移だけでなく、バグの内容に着目することで、より詳細に分析することが可能になります。ここでは、どんなバグか、いつ発見されたか、いつ混入したか、なぜ発見できなかったのか、なぜ作り込んだのかなどをパレート図やクロス分析を利用して分析します。ここでの分析は、表4.1に示した管理項目のうち、以下の項目を利用して行います。

システム全体の分析で使用した事例（図6.3）において、バグの内容まで分析した事例を説明します。今回はシステム全体に対して分析していますが、サブシステムを考慮することでより良い分析になります。

●利用するバグ管理項目

No	バグ管理項目	利用目的
1	管理番号	バグの件数を計測するため
2	重要度	重要度別に計測するため
3	発見工程	いつバグが発見されたかを計測するため
4	作り込み工程	いつバグが作り込まれたかを計測するため
5	バグ区分	バグの原因を計測するため
6	発見すべき工程	本来、バグを発見すべき工程を計測するため
7	発見すべきアクティビティ	本来、バグを発見すべきアクティビティを計測するため

以下の分析事例は上記のバグ管理項目の一部を利用したものです。他の管理項目も利用することで、より詳細な分析が可能になります。例えば、優先度やステータス、発生原因などについても併せて分析をすると効果があると考えられます。

●重要度の分析

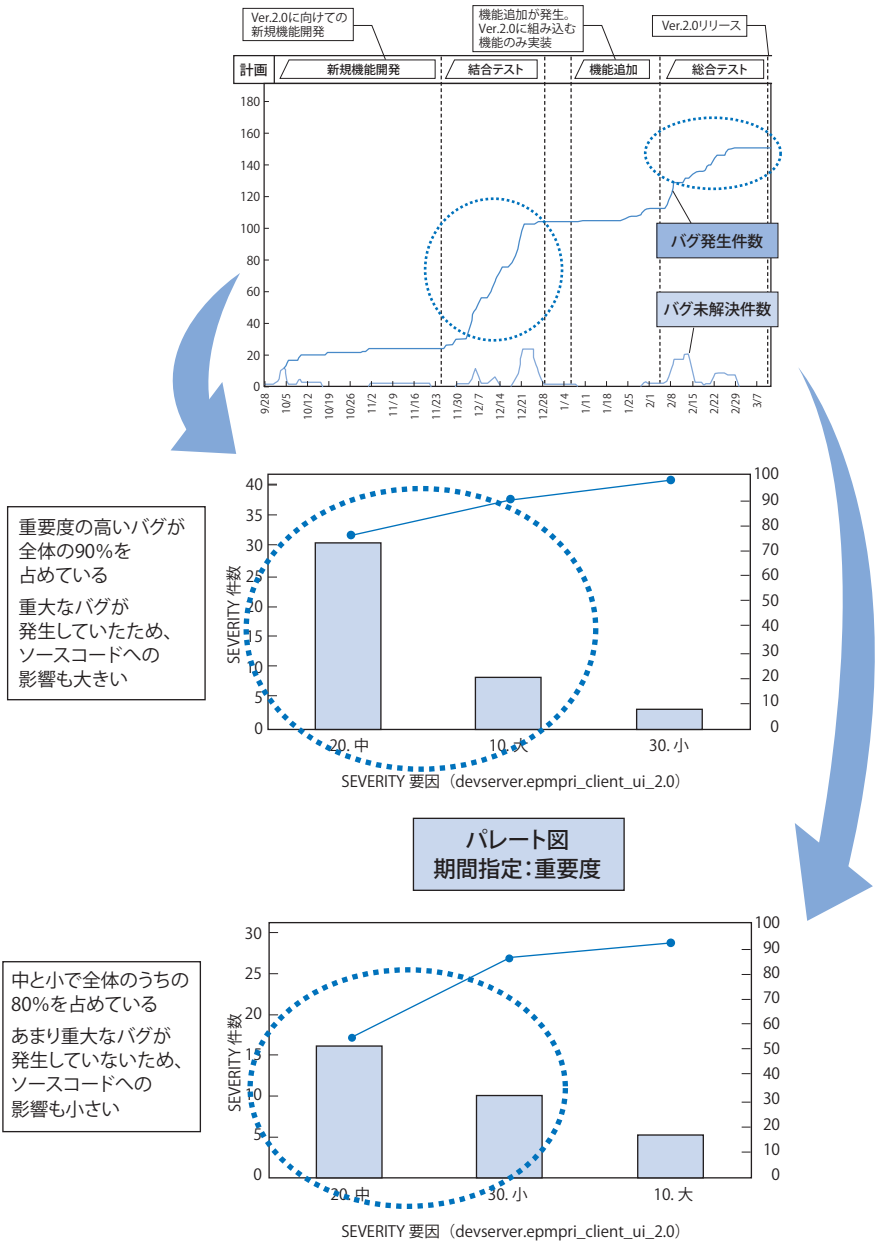


図 6.5　バグ内容の分析（パレート図による重要度分析）

前頁図 6.5 で、点線で囲った箇所の変化点に着目した分析を紹介します。

- 結合テストの期間中に発生したバグは、全体の 90%を重要度の高いものが占めていることが分かります。重要度の高いバグが発生しているため、ソースコードへの影響も大きく品質にも影響するため、再テストが必要になります。
- 総合テストの期間中に発生したバグは、重要度が中と小のもので全体の 80%を占めていることが分かります。あまり重要度の高いバグが発生していないため、ソースコードへの影響も小さいと考えられます。

●混入工程と発見工程の分析

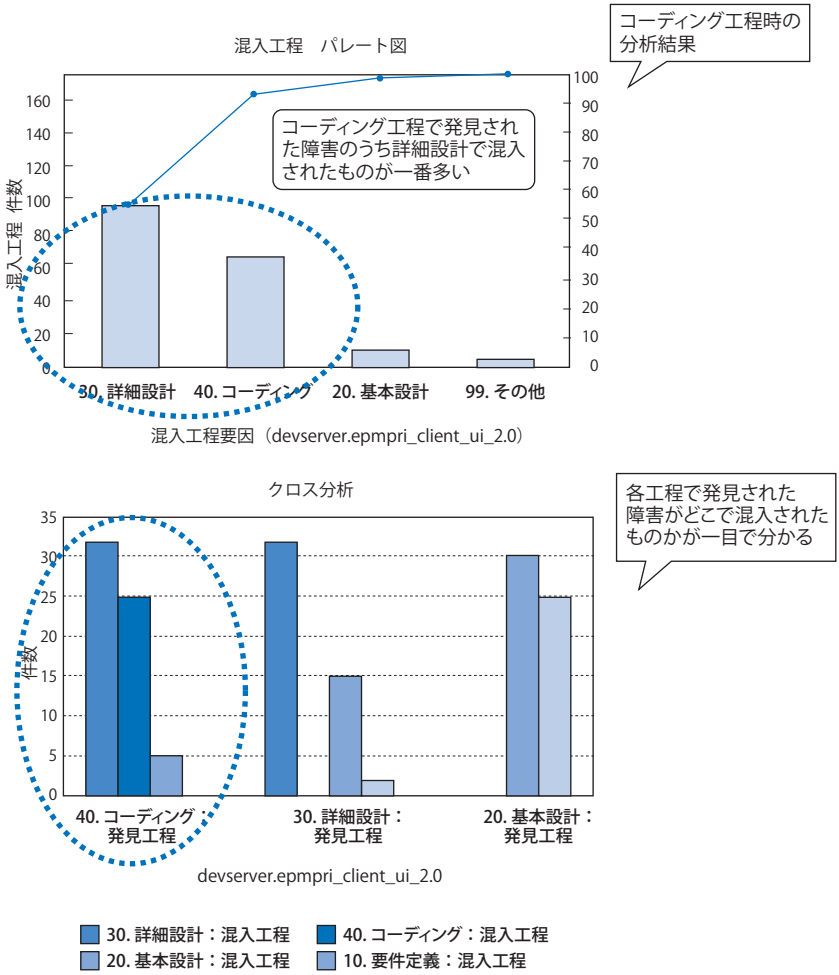


図 6.6　バグ内容の分析（混入・発見工程分析 1）

前頁図 6.6 で、点線で囲った箇所の変化点に着目した分析を紹介します。

- 上のパレート図と下のクロス分析から、実装工程（コーディング／単体テスト）で発見されたバグのうち詳細設計で混入されたものが最も多いことが分かります。本来発見すべき工程から遅れれば遅れるほど、発見されたバグに対してその原因の特定から対策の検証までに膨大な手戻り工数が必要になります。

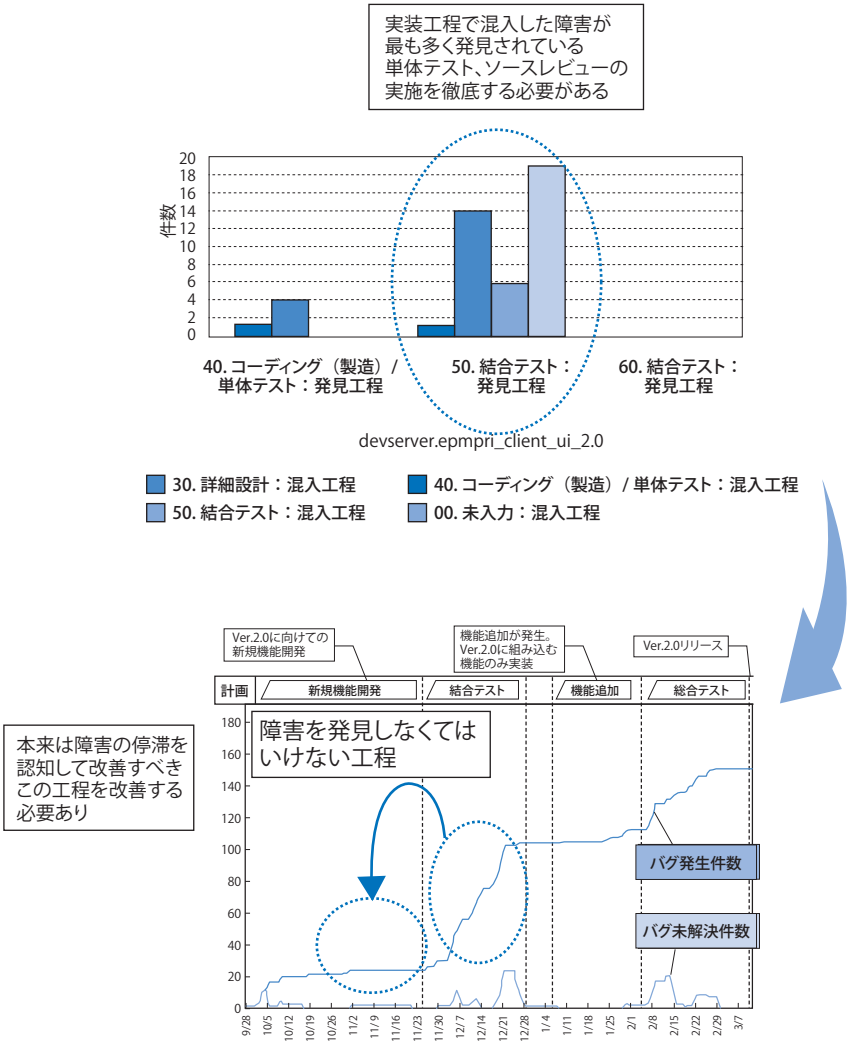


図 6.7　バグ内容の分析（混入・発見工程分析 2）

前頁図 6.7 で、点線で囲った箇所の変化点に着目した分析を紹介します。

- 前頁上図のクロス分析（混入・発見工程）から分かるように、結合テストで発見されたバグのうち実装工程で混入したものが最も多く発見されていることが分かります。これは図 6.3 の結合テスト工程でバグの発生件数が停滞していることとも一致しているものです。

Column

「なぜなぜ分析」
～バグの根本原因の追究手段～

バグは重大度によって、対処（処置、現象の解消）に加えて対策（改善、原因の除去）が必要になります。効果的な対策を立案・実施するために、問題の根本的な原因を明確にすることが重要です。ここでは具体例をもとに、根本原因の追究の手段である「なぜなぜ分析」を紹介します。

●「なぜなぜ分析」の具体例

以下のバグ事象とその直接原因に対して、「なぜなぜ分析」を使用して根本原因を明確にする分析プロセスを示します。

バグ事象：○○装置にて情報登録オペレーションを行うと、装置カウンタが＋1されるところ、＋2されてしまう。

直接原因：装置カウンタに設定される値はワークカウンタ（共通部品）が保持している。ワークカウンタ（共通部品）の値が情報登録オペレーション前の“既存情報”の表示の際に＋1され、さらに情報登録オペレーションの際に＋1されており、カウンタの二重更新となっていた。

●「なぜなぜ分析」の例

「なぜなぜ分析」の例を右に示します。

●「なぜなぜ分析」から導き出す対策例

- **短期対策**

・内部仕様の変更に対する記録の見直しを実施する
・ワークカウンタ（共通部品）が内部仕様通りに動作するかの観点で強化テストを行う

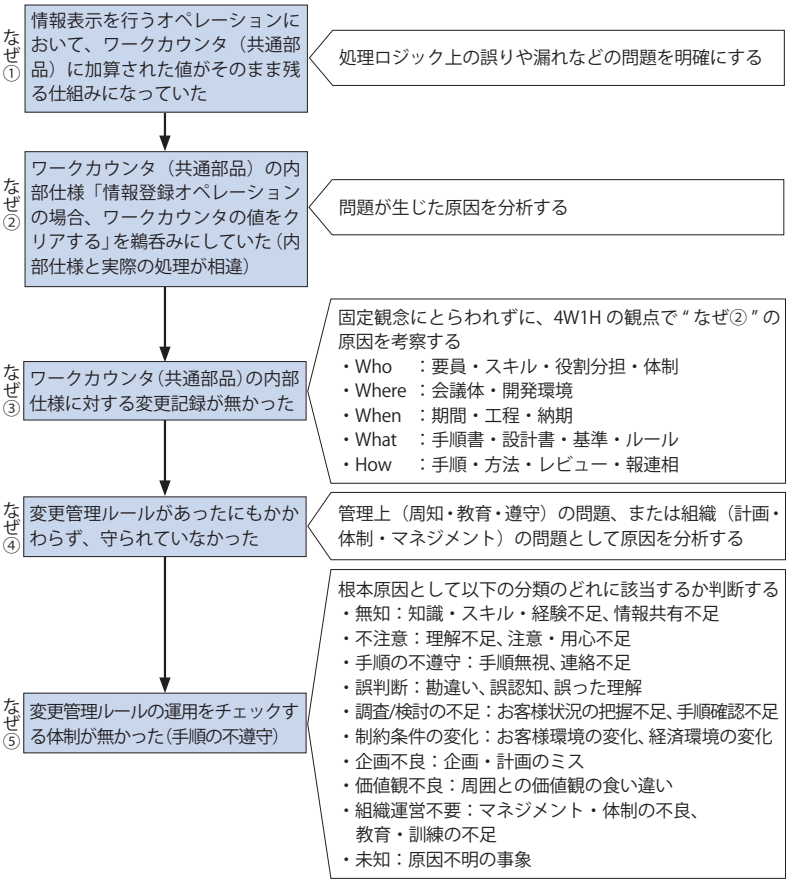
考え方	・「なぜなぜ分析」“なぜ①～③”に対する対策を立案する ・対策の実施対象は、担当者・関係者の範囲とする

- **長期対策**

・変更管理ルールを開発メンバ全員に改めて周知する
　具体的には、チーム単位で説明会を実施する
・体制・役割分担を見直して変更管理チームを組織し、変更管理をルール通りに運用する

考え方
・「なぜなぜ分析」“なぜ④～⑤”に対する対策を立案する
・対策の実施対象は、プロジェクト（開発チーム）全体とする

「なぜなぜ分析」の例



（3）バグ発生件数とソースコード行数の組み合わせ分析

バグの発生件数とソースコード行数を組み合わせた場合の事例を紹介します。ここではバグの発生状況だけでなく、ソースコードの開発状況を考慮した品質の分析を実施します。今回はシステム全体に対して分析していますが、サブシステムを考慮することでより良い分析になります。

●利用するバグ管理項目

No	バグ管理項目	利用目的
1	管理番号	バグの件数を計測するため
2	発行日／発見日時	バグがいつ発生したかを計測するため（発生件数）
3	処置日／完了日	バグがいつ解決したかを計測するため（未解決件数）
4	機能名／サブシステム名	サブシステムごとに計測するため

バグ管理項目ではありませんが、テスト対象のソースコード行数も計測して、バグ管理項目と合わせて分析します。

以下、事例をもとに説明します。

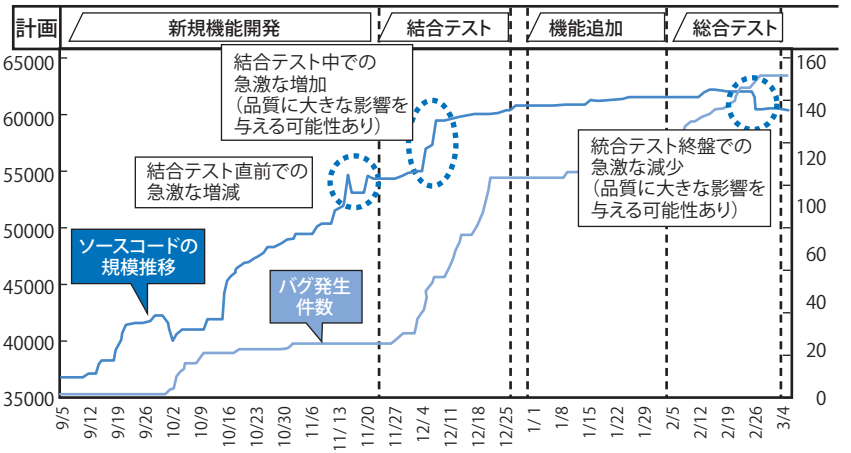


図 6.8　バグ発生件数・ソースコード行数分析

前頁図 6.8 で、点線で囲った箇所に着目した分析を紹介します。

- 結合テストの直前でソースコードの急激な増減が発生しています。単体テストが実施されていないソースコードが存在する可能性があるため、結合テストで品質に問題が発生する可能性が高いと考えられます。
- 結合テスト中にソースコードの急激な増加が発生しています。結合テスト中に機能追加されている可能性を示しており、テスト計画を見直すなどの対応が必要になります。
- 総合テスト終了間際でソースコードが急激に減少しています。このことは品質に大きな影響を与える可能性があります。リリース間近のため、ソースコードが削除された状態でシステムが正しく動作するかの確認が必要になります。

（4）バグの発生件数とテストケース数の組み合わせ分析

バグの発生件数とテストケース数を組み合わせた場合の事例を紹介します。

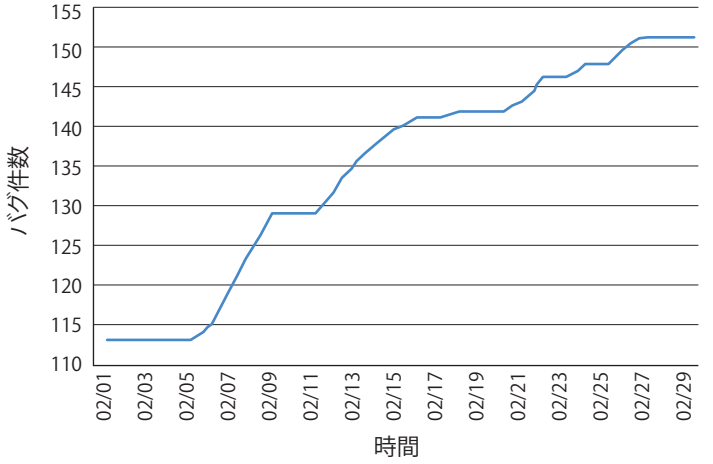
バグの収束の分析では、テストケースの消化が経過時間と比例しない場合、テストケース数の消化状況を考慮した分析を実施する必要があります。

●利用するバグ管理項目

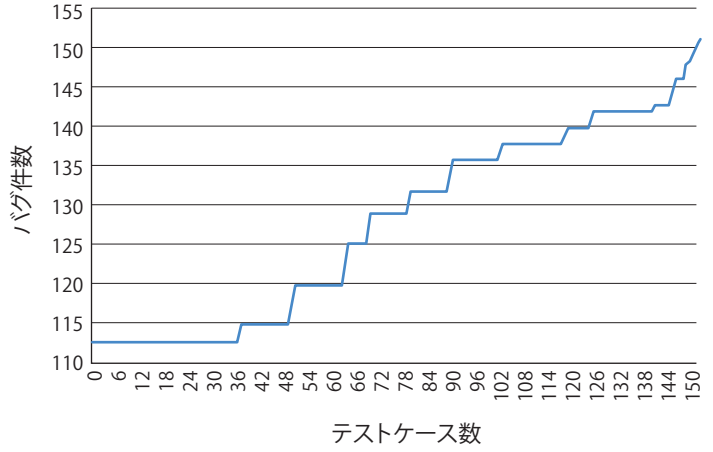
No	バグ管理項目	利用目的
1	管理番号	バグの件数を計測するため
2	発行日／発見日時	バグがいつ発生したかを計測するため（発生件数）
3	処置日／完了日	バグがいつ解決したかを計測するため（未解決件数）

バグ管理項目ではありませんが、テスト対象のテストケース数とテストケースの実施件数も計測してバグ管理項目と併せて分析します。

まず、バグの発生件数のグラフのX軸をそれぞれ「時間」と「テストケース数」にした場合の例を比較して、テストケース数を用いない場合の問題を説明します。



X軸を「時間」とした場合　→　収束しているように見える

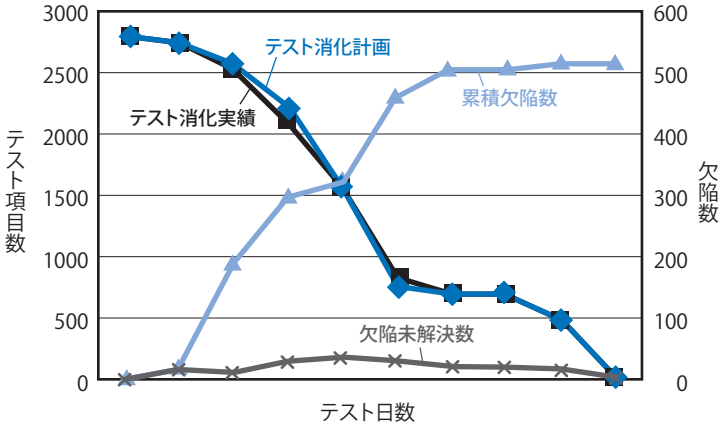


X軸を「テストケース数」とした場合　→　収束していない

図 6.9　テストケース分析

図 6.9 のグラフからは、X 軸に「時間」を設定した場合にはバグが収束しているように見えますが、「テストケース数」を設定した場合には収束していないことが分かります。X 軸に時間を設定した際には、テストを実施しなければ収束してしまうため、正しくバグの収束を予測することができません。

そこで、テストケースの消化実績・消化計画やバグの発生件数・未解決件数のメトリックスを収集している場合には、「定量的品質予測のススメ」60 頁で紹介されている「品質評価グラフ」が良いと考えられます（図 6.10 参照）。



出典：「定量的品質予測のススメ」（IPA/SEC）をもとに作成

図 6.10　品質評価グラフ

Column

バグの収束とバグ曲線

バグの収束の確認で最も有名な手法は、バグ曲線（正式には信頼度成長曲線）です。

バグ曲線とは簡単にいうと、横軸に実施したテストの項目数または時間、縦軸に検出したバグ数をとったグラフです。バグの検出数が実施したテスト項目数に関連していることから、その関連を統計的にモデル化して予測し、実際の結果と比較することで収束を判断します。モデル、すなわち曲線の形が予測できれば、その予測と実際の結果を比較することでバグの収束を判断することができるという考え方です。

バグ曲線の詳細については、「定量的品質予測のススメ」（IPA/SEC、株式会社オーム社、2008年）（pp.91-95 Annex B ソフトウェア信頼度成長モデル）を参照してください。

●バグ曲線の課題

バグ曲線は、バグの収束を検討する上で多くのプロジェクトが参考にしている手法です。しかし残念ながら、その結果だけで実際にバグが収束しているかどうかを判断できることはあまり多くありません。

この曲線を利用するためには、まず以下の点に注意することが必要です。

- テスト項目に偏りがなく、必要にして十分な項目が実施されていること。
- 横軸のテスト項目数の粒度が揃っており、実施した項目数をきちんとプロットできること（項目1件の粒度が違いすぎると、横軸をテスト項目数としても無意味になる）。

テスト項目の必要十分性や項目の粒度をどのように考えればよいかは、いずれも難しい問題です。現時点では明確な考え方が示されているとは言いがたい状況です。これだけをみても、バグ曲線をバグ収束に用いる難しさを理解していただけるのではないでしょうか。なお、テスト項目の必要十分性や粒度については、「組込みソフトウェア開発

における品質向上の勧め［テスト編～事例集～］」（IPA/SEC、2012 年）に説明がありますので参考にしてください。

さらに、バグ曲線の形を予測するモデルの選択も難しい問題です。モデルは、ゴンペルツ曲線や指数形モデル、遅延 S 字形モデルなど、数多くが提案されています。これらの中から、どれを選択すればよいのかが難問です。モデルはプロジェクトにあわせたテーラリングも推奨されていますが、テーラリングとなると、また一段と敷居が高くなります。

●バグ曲線の課題への対応

バグ曲線には様々な課題がありますが、それでも多くの組織が上記の課題に現実的に対処して、この曲線をバグの収束の参考として利用しています。

例えば、以下のような対応方法が報告されています。

- 横軸には、日付をとる。
  バグ曲線をあくまで参考として利用する場合、横軸を日付としても十分参考になります。ただし、日付とテスト項目の消化状況の関係には十分な注意が必要です。
- モデルとする曲線の選択は、3 種類程度の曲線を対象として、実際の結果の曲線を比較し最も近いモデルを選択する。

いずれにしても、バグの収束の判断では、バグ曲線だけではなく実際に発生したバグの分析などその他の多くの情報をもとに行うことが必須です。

（5）バグ抽出状況分析

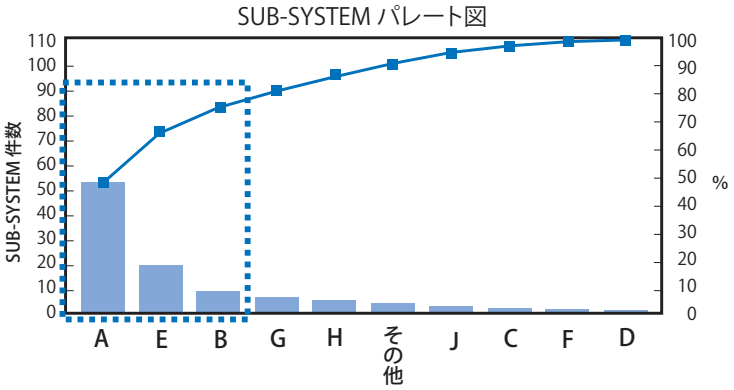
バグの発生件数とテストケース数を測定した場合の事例を紹介します。時系列に並べた推移を分析するだけでなく、異なるメトリックスの件数や規模を組み合わせることでより詳細な分析ができます。

●利用するバグ管理項目

No	バグ管理項目	利用目的
1	管理番号	バグの件数を計測するため
2	発行日／発見日時	バグがいつ発生したかを計測するため（発生件数）
3	処置日／完了日	バグがいつ解決したかを計測するため（未解決件数）
4	機能名／サブシステム名	サブシステムごとに計測するため

バグ管理項目ではありませんが、テスト対象のテストケース数、テストケースの実施件数も計測して、バグ管理項目と併せて分析します。

以下、バグ密度やテスト密度を考慮した分析事例をもとに説明します。



累積障害件数／実施件数

	実施件数	累積障害件数	障害発生率	課題件数
A	744	55	7.39%	10
B	113	10	8.85%	0
C	200	3	1.50%	0
D	124	2	1.61%	0
E	365	20	5.48%	1
F	21	2	9.52%	0
G	148	9	6.08%	1
H	98	8	8.16%	0
I	25	0	0.00%	0
J	60	4	6.67%	1
K	14	0	0.00%	1
合計	1912	113	5.91%	14

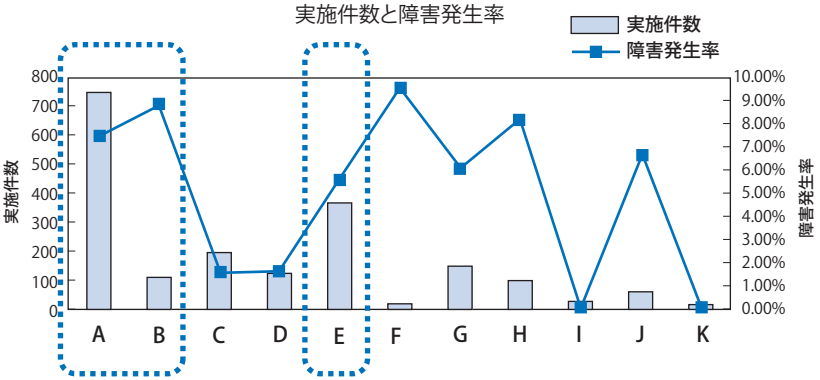


図 6.11　バグ抽出状況分析

前頁図 6.11 で、点線で囲った箇所に着目した分析を紹介します。

- 上段図のサブシステム単位でのパレート図の分析から、機能 A、E、B にバグが多く発生していることが分かります。
- 中段表、下段図のバグ発生率（バグ発生件数／テストケース実施件数）から、問題となり得る機能 A、E、B の中で、B は最もバグ発生件数が少ないがバグ発生率が最も高くなっていることが分かります。このことから、テストケースを追加することで、残存バグを発見することができる可能性があります（本分析を実施する際にサブシステムごとの複雑度が分かると、より良い分析になります）。

参考文献・ウェブサイト

■ JIS X 0133-1:1999 ソフトウェア製品の評価

■ JIS X 0014:1999 情報処理用語－信頼性、保守性及び可用性

■ JIS Q 9000:2006 品質マネジメントシステム－基本及び用語

■ JIS C 0508-2:2012 電気・電子・プログラマブル電子化安全関係の機能安全

■ JIS Z 8115:2000 ディペンダビリティ（信頼性）用語

■ソフトウェアテスト標準用語集Ⅴ 2.0（日本語版）

■ IEEE Std 982.1-2005 IEEE Standard Dictionary of Measures of the Software Aspects of Dependability, 2005

■ IEEE Std 1044.1-1995 IEEE Guide to Classification for Software Anomalies, Institute of Electrical and Electronics Engineers, 1996

■ IEEE Std 1044-2009 IEEE Standard Classification for Software Anomalies, Institute of Electrical and Electronics Engineers, 1992

■ SEC BOOKS「組込みソフトウェア開発における品質向上の勧め［テスト編～事例集～］」IPA/SEC、2012 年

■ SEC BOOKS「【改訂版】組込みソフトウェア開発向け品質作り込みガイド」IPA/SEC、2012 年

■ SEC BOOKS「定量的品質予測のススメ～ IT システム開発における品質予測の実践的アプローチ～」IPA/SEC、株式会社オーム社、2008 年

■ SEC BOOKS「【改訂版】組込みソフトウェア向け開発プロセスガイド」IPA/SEC、翔泳社、2007 年

■ Mantis Bug Tracker, http://www.mantisbt.org/

■ Bugzilla, http://www.bugzilla.org/

■ Software Quality Measurement: A Framework for Counting Problems and Defects, William A. Florac, Technical Report CMU/SEI-92-TR-022, 1992

■ New Ways to Get Accurate Reliability Measures, Sarah Brocklehurst, Bev Littlewood, Journal IEEE Software, Volume 9 Issue 4, 1992

■野中誠、小池利和、小室睦「データ指向のソフトウェア品質マネジメント～メトリクス分析による『事実にもとづく管理』の実践」、日科技連出版社、2012 年

執筆者（敬称略・50 音順）

石尾　渉　　オムロン オートモーティブエレクトロニクス株式会社
植武　信弘　　株式会社日立情報制御ソリューションズ
岡本　孝之　　株式会社富士通ソフトウェアテクノロジーズ
千葉　正史　　三菱電機株式会社
十山　圭介　　IPA/SEC（株式会社日立製作所）
西野　敦士　　ルネサス エレクトロニクス株式会社
三橋　二彩子　日本電気株式会社
三原　幸博　　IPA/SEC 組込み系プロジェクト　プロジェクト・リーダー
（アルパイン株式会社）
宮本　貴之　　一般社団法人 TERAS（キャッツ株式会社）
村山　耕一　　イーソル株式会社
山上　宣彦　　東芝ソリューション株式会社

監　修

バグ管理手法部会

SEC BOOKS

組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〕

2013年3月8日　1版1刷発行

監 修 者　独立行政法人情報処理推進機構（IPA）
技術本部 ソフトウェア・エンジニアリング・センター（SEC）

発 行 人　松本　隆明

発 行 所　独立行政法人情報処理推進機構（IPA）
〒113-6591
東京都文京区本駒込二丁目 28 番 8 号
文京グリーンコート センターオフィス
URL　http://sec.ipa.go.jp/

ISBN978-4-905318-16-3　Printed in Japan