5．バグの追跡

|  |
| --- |
| The most prominent place in hell is reserved for those who are neutral on the great issues of life.  人生の重要な課題を放置する人は、地獄でも屈指の場所に送られるだろう。  Billy Graham (アメリカの伝道師) |

# バグの追跡

バグとは、ソフトウェアが意図どおりに動かない不具合のことです。ソフトウェアの開発中は、山のようにバグが出てきます。製品をユーザーにリリースする前までに、これらを漏れなく解決(修正)しなければなりません。そのためには、個々のバグの状況を、最後まで正しく追跡する必要があります。ちゃんと治ったかどうか、追いかけていくということです。バグの追跡プロセスはソフトウェア開発の下流工程では非常に重要なもので、それはそのままソフトウェア開発の下流工程を規定するものといっても過言ではありません。バグは、開発者とテスターの間で、バグ報告票 (Bug Report) をやり取りしながら追跡します。バグ報告票は、開発規模によってはすぐに数百から数千といった数になるので、紙に書いても管理できません。そこで、Microsoft Excelのようなスプレッドシートや、バグ追跡システム(BTS; Bug Tracking System)などのソフトウェアを使います。スプレッドシートよりもBTSを使う方がずっと便利ですが、まずはソフトウェアやツールに依存しない形で、バグを追跡するための一般的な原則について学びましょう。

# バグ報告票のライフサイクル

バグを発見したら、ひとつのバグにつき、ひとつのバグ報告票を起票します。このバグ報告票のライフサイクルを、図5 - 1に示します。これは、バグ報告票の状態遷移をUMLの状態遷移図で表したものです。



図5 - 1 バグ報告票の状態遷移

バグ報告票を新しく起票するのは、誰がしても構いません。しかし、バグ報告票がどうあるべきかを理解していない人 (プロのテスターではない人) が記述した場合、そのバグ報告票の品質に問題が出ることがあります。このため、プロジェクトによっては、バグ報告票を起票できる人を テスターだけに制限することもあります。

バグ報告票を起票したら、この修正に責任をもつ開発者と、その修正の確認に責任をもつテスターの両方を最初に割り当てます。つまり、このバグ報告票の ”開発(修正)担当者” と “テスト担当者” の項目に、それぞれ担当者名を記入します。そして、このバグ報告票の状態によって、その<★傍点2コ/>の担当者が決まります。バグがまだ治っていないときは開発者が修整を担当し、バグが治った後はテスターがテストを担当する、というわけです。これを表5 - 1にまとめました。バグ報告票の状態についても、各バグ報告票の ”状態” の項目に記載して管理します。バグ報告票の状態遷移に伴い、現在の担当者も遷移します。ただし、バグ報告票の状態遷移を経ずに、現在の担当者を一時的に変更することもあるので、現在の担当者を記入する項目もバグ報告票に準備します。このほか、フィーチャーオーナーやプロジェクトリードの名前を記入する項目を用意することもあります。

表5 - 1 バグ報告票の状態と担当者

|  |  |  |
| --- | --- | --- |
| 状態 | 状態の別名 | 担当者 |
| 新規 | New, Triage Neededなど | プロジェクトリード、  デブリード、  トリアージチームなど |
| 割り当て済み | 開始, 開始済み, 着手済み, Open, Active, Assigned, Started, Reproducedなど | 開発者 |
| 解決済み | Resolved, Pendingなど | テスター |
| 確認済み | 検証済み, Verifiedなど | デブリード、 テストリードなど |
| 終了 | 済み, 完了, Closed, Doneなど | (なし) |

バグ報告票の状態遷移を、担当者間のインタラクションとしてシーケンス図で表したものを図5 - 2に示します。この図では、担当者をアサインするのはプロジェクトリードとしていますが、これをデブリードもしくはトリアージチームの責務とすることもあります。



図5 - 2 バグ報告票を閉じる手順

このように、バグ報告票はそれぞれ状態をもちます。最初に記述されたときの状態は［新規］です。以下に、各状態でどのようなアクションが必要になるのかを説明します。

|  |
| --- |
| バグ報告票の状態と処理方法の区別はとても重要です。そこで、本書では状態を［大かっこ］で、処理方法を「かぎかっこ」で区別して表示しました。 |

## ［新規］(New)

担当者はプロジェクトリードです。このバグを開発者にアサインする (割り当てる) 前に、記述されたバグ報告票の妥当性を検査し、このバグにしかるべき開発者を割り当てます。また、前述の通り、この報告票が［解決済み］に遷移したとき、その解決を確認する担当者 (テスター) をこの段階で同時に割り当てます。テスターを割り当てる代わりに、この報告票の起票者が確認を担当する運用もあります。このバグを報告した人が、最も適切にこのバグの解決(修正)を確認できると期待できるからです。

ユーザーからバグの報告があったときなど、バグを発見した人とバグ報告票を起票する人が異なることもあります。そのような場合は、社内のテスターがバグの再現を確認してからバグ報告票を起票します。あるいは、忘れないようにまず起票してから、テスターが再現を試行する運用もあります。みなさんの組織にあった起票ポリシーや、担当者の割り当てポリシーを探してみてください。

## ［割り当て済み］(Open)

担当者は開発者です。バグを解決する方法(処理方法)はいくつかあります。開発者は、複数の処理方法の中から1つを選択し、実際にそのバグに対するアクションを実施したうえで、バグ報告票の状態を［解決済み］とします。例えば、このバグを修正し、修正ファイルを3章に示した手順でリポジトリにコミットしたら、バグ報告票を「修正済み」という処理方法で解決します。このとき、コミット時のチェンジセット番号 (コミット番号) と、その修正が反映される次のビルド番号をバグ報告票に追記し、バグ報告票の状態を［解決済み］とします。ここで開発者が取れる「修正済み」以外のアクション(処理方法)については後述します。

バグ報告票に最初にアサインされた開発者が、この担当者には不適格であることが後で判明することもあります。この場合には、その開発者自身もしくは開発マネージャにより、違う開発者が担当者としてリアサインされます。これは最初の担当者が能力的に不適格であったというわけではなく(まれにそういう場合もありますが)、バグの原因であるサブシステムを読み違えたことによるものです。最初にアサインされた開発者は、バグに関する調査結果を追記して、次の担当者にこのバグ報告票を受け渡します(“開発担当”の項目にある自分の名前を消して、次の担当者の名前を記入するということです)。あるいは、特定の開発者の負荷が重過ぎるので、のために開発者をアサインし直すこともあります。また、いちど閉じられたバグ報告票を再開して、この状態にまで差し戻したときも、最初に担当した開発者とは違う人をアサインすることがあります。

## ［解決済み］(Resolved)

担当者はテスターです。担当者は、開発者が選択した処理方法に同意できるか確認し、問題なければバグ報告票の状態を［確認済み］にします。例えば、開発者が「修正済み」という処理方法でバグ報告票を［解決済み］にしたとき、その時点ではその修正を含むビルドはまだテストチームにリリースされていないでしょう。しかし、状態が［解決済み］に変わることで、このバグ報告票はテスターにアサインされ、テストが保留中の状態になります。テスターは、修正の確認に必要な手順を確認したり、テストケースを追加するなどして準備します。バグ報告票に記載のある番号のビルドがリリースされたら、このビルド(もしくは、これより大きな番号のビルド)をテストして当該のバグが正しく修正されたどうかを確認します。修正されていれば、テストに使ったビルドの番号をこのバグ報告票に記入して、その状態を［確認済み］に変更します。プロジェクトによっては、［確認済み］という状態を設けず、テスターが直接［終了］という状態にすることもあります。修正されていなかったなど、開発者のバグの処理方法にテスターが同意できなければ、バグ報告票の状態を［割り当て済み］に戻して、これを開発者に差し戻します。

## ［確認済み］(Verified)

担当者はプロジェクトリードです。「修正済み」という形で解決されているものは特に問題になりません。担当者は、念のため、かるくバグ報告票の内容を確認したうえで、これを［終了］という状態にします。「バグではない」「修正する予定はない」といった形で［確認済み］になっているバグは、担当者は本当にそれで問題ないか吟味したうえで、［終了］という状態にします。このとき、担当者が処理方法に問題があると判断すれば、その理由をバグ報告票に追記したうえで、これを［割り当て済み］の状態にまで差し戻します。

## ［終了］(Closed)

担当者はいません。これで最後です。ソフトウェアをリリースするときには、どんな形であれ、すべてのバグ報告票をクローズしなければいけません。また、各マイルストーンをexitするときにも、そのマイルストーンに割り当てられたバグ報告票はすべてクローズすべきです。ただし、［終了］となったバグ報告票でも、これは各バグの履歴を記録・追跡するための重要な資料ですから、破棄してはいけません[[1]](#footnote-1)。例えば、［終了］となったバグ報告票でも、その再現手順はリグレッションテストのインプットとして利用されます。また、バグ報告票データベースはユーザーサポートのためのナレッジベースとしても活用されます。バグ報告票は、ソフトウェア開発ではコストをかけて記述・保守すべき重要な資産なのです。

# バグ報告票の流通

一つひとつのバグ報告票が、このような状態遷移を経て解決され、クローズされていきます。バグの追跡プロセスは、このワークフローを定義します。このワークフローにのって、非常にたくさんのバグ報告票がチーム内を流通します。この流通量で、現在のプロジェクトの状態を測ったり、各開発者やテスターの負荷が適正に分散されているかを確認したりできます。

また、滞っているバグ報告票がないかどうか監視することも大切です。例えば、担当者名を含む任意の項目の更新が1週間以上止まっているバグ報告票は、その担当者のところで滞っています。このバグ報告票の処理をブロックしている原因を探して、取り除かねばなりません。バグ報告票の解決が滞ると、プロジェクト自体の進捗も滞ってしまうからです。



図5 - 3 バグ報告票がチーム内を流通しながら  
解決され、閉じられていくようす

# バグの発見と文書化

一般に、発見したら起票の必要があるバグは、きちんとビルド番号が管理された (つまりビルドマシンでビルドされ、テストチームにリリースされた) 統合ビルドで再現したバグだけです。開発者がサンドボックスでのビルドで見つけたバグは、バグ報告票を起票する必要はありません。そのバグは、その開発者のサンドボックスの汚れたファイル(修正してまだコミットしていないファイル)が原因かもしれないからです。開発者はその問題を起票することなく、こっそり治してしまえばOKです。ただし、このバグを含むファイルがすでにリポジトリにコミットされていて、ビルドマシンでつくった統合ビルドにおいてもこのバグが再現すると考えられるときは、バグ報告票の起票が必要です。そのバグがビルドマシンでビルドした統合ビルドでも再現することを確認したうえで、開発者が自分で (あるいはテスターに依頼して) バグ報告票を起票します。

将来のビルドで実装予定の機能や、既知のバグについては、テストチームに無駄な作業をさせないために、制限事項として開発者がリリースノートに記述します。また、この制限事項を解除するための作業をタスクとしてバグ報告票に起票し、自分にアサインしておくと便利です(あるいは、このようなタスクの起票を義務づける運用もあります)。

# 一件一葉の原則

ひとつのバグにつき、ひとつのバグ報告票を起票します。これを一件一葉の原則といいます[[2]](#footnote-2)。

例えば、開発中のソフトウェアのある現象の中に複数のバグが含まれていることがありますが、その場合にはそれぞれのバグについて別のバグ報告票を起票すべきです。そうでなければ、開発者はバグ報告票に含まれるすべてのバグを同時に修正しないと、そのバグ報告票を処理できず、作業が混乱するからです。また、バグ報告票の数はソフトウェアの品質を測るための重要な基準を提供しますが、一件一葉を守らないと、この数字も不正確になってしまいます。

逆に、すでにバグ報告票を起票済みのバグについて、誤って別のバグ報告票を起票してしまうこともありますが、これも好ましくありません。これを避けるには、バグ報告票を起票する前にバグ報告票データベースを検索して、報告したいバグがすでに起票済みかどうか探してみることです。もし見つかれば、内容を確認し、追記できることがあれば追記します。誤って重複したバグ報告票を起票したときの処理については後述します。

ただし、さくらんぼ摘みポートが計画された、親子関係にある別のブランチに同じバグを見つけたときは、ブランチの数だけバグ報告票を起票する方が便利です。3-27節でも説明しましたが、一方のブランチでバグを修正してバグ報告票を解決しても、もう一方のブランチに関連づけられたバグ報告票はオープンのままにしておけるからです。バグの起票ポリシーについては、3-31節も参照してください。

# コラム ソフトウェアの不具合をバグというのはなぜ？

|  |
| --- |
| **コラム ソフトウェアの不具合をバグというのはなぜ？**  ソフトウェアの不具合のことを、英語ではerrorとかbugなどといいます。エラーはソフトウェア以外の文脈でも、普通に失敗や間違いを指す言葉です。例えば、野球でも失策や捕球ミスをエラーといいますね。しかし、ふつうバグといえば昆虫のことです。なぜソフトウェアの不具合をバグというのでしょうか。  1940年代から50年代の頃のコンピュータは、半導体ではなく真空管でできていました。この真空管から漏れる光と熱に誘われた虫がコンピュータの高圧電流で焼死し、その死骸がコンピュータの不正な動作を引き起こすことが実際にありました。そこで、ソフトウェアの不具合を指してバグというようになりました。また、この虫を取り除いてコンピュータが動作するように治したことから、ソフトウェアの不具合を除去する作業を指してデバッグ (虫の除去) というようになりました。  一般に、昆虫はinsectといいますが、bugといえばその中でもてんとう虫のような甲虫を想起させます。このため、IDEのデバッグボタンには、よくてんとう虫のアイコンがあしらわれます。  このほかのバグの別名には、障害、欠陥、故障、瑕疵、不備、仕様[[3]](#footnote-3)などがあります。 |

# コラム ソフトウェア開発のメタファ⑤ 医療

|  |
| --- |
| **コラム ソフトウェア開発のメタファ⑤ 医療**  ソフトウェアのバグとは、患者の病気やけがのようなものです。本書では繰り返し述べていることですが、ソフトウェア開発を行う際には、その健康を維持することがとても重要です。実際に、ソフトウェア開発には、医療で使われる用語や概念を取り入れたものがあります。例えば、BVT (Build Verification Test; ビルドをテストチームにリリースする前に実施するテスト) などのテストを、health check (健康診断) ということがあります。定期的にビルドの健康診断をし、その状態を観察することが、ソフトウェアの進化にはとても大切なのです。また、本章では「トリアージ」という医療とソフトウェアの両方で使われる用語を紹介します。このほか、6章ではソフトウェア開発におけるポストモータム (検死) というプラクティスも紹介します。さらに、緊急を要する救急患者の移送プロセスを参考にした、緊急を要するソフトウェアの補修プロセスや、品質バイタルサイン(quality vital sign)など、医療にヒントを得たソフトウェア開発・保守の話題があります。2010年11月にアメリカのラスベガスで開かれたIBM Information On Demand Conferenceでは、外科医が招かれて基調講演を行っています。  また、フレデリック・ ブルックス氏の著書「人月の神話」は、ソフトウェア開発プロジェクトチームを外科手術チームにたとえて、より大勢のメンバーをうまく機能させる方法について述べています。人月 (man-month)とは一人がひと月に働ける作業量のことで、ソフトウェア開発の見積もりによく使われる単位です。しかし、例えば36人月とは何を意味するのでしょうか。1人が3年かかって作るものを、36人が1ヶ月で作れるはずはありません。このように、「人月の神話」は人と月が交換可能であるというのは神話 (迷信) だとしています。しかし、大規模なソフトウェアを短期間に作るためには、ある程度の人員を投入してうまく機能させることが必要であり、複数の役割から構成される外科手術チームとソフトウェア開発チームを対比させているのです。この本は、「狼人間を撃つ銀の弾丸はない」(ソフトウェア開発の問題をすべて解決する便利な方法などない)という名句を残していることでも有名ですね。  ところで、これだけ医療とソフトウェアに類似性がある中で、ソフトウェアの「移植」を英語では port ということはなんだか不思議です。人体の臓器移植は、英語では「organ transplant」といい、port という語は使いません。この語の由来は、portable (持ち運びできる) から来ているのでしょうが、ソフトウェア開発の文脈におけるportを「移植」と最初に訳した人のセンスは素晴らしいと思います。 |

# バグの処理方法 (Resolution)

バグ報告票の状態が［割り当て済み］から［解決済み］に遷移するときに、開発者が取れるアクションのことを処理方法 (もしくは対処方法など) といいます。よく使われる「処理方法」について、以下に紹介しておきます。これらは、先に紹介したバグ報告票の［状態］とはまったく別のものなので注意してください。開発するソフトウェアのドメインによっては、ここに紹介した以外の処理方法も用意しておくとよいでしょう。

表5 - 2 バグの処理方法と別名 (例)



## 「修正済み」 (Fixed)

バグの報告者が、最も期待する処理方法です。このとき、開発者は修正が反映されるビルドの番号や、どのソースコードを修正したか、この修正をリポジトリにコミットしたときの番号 (チェンジセット番号) などをバグ報告票に追記します。また、より簡単な再現方法があれば、それも追記します。テスターは、意図したとおりに修正されていることを当該のビルドで確認・検証します。修正を確認できれば、テスト結果をこのバグ報告票に追記し、バグ報告票の状態を［確認済み］とします。もしも、テスターがバグの再現手順について不安があれば、このバグを報告した人にアドバイスを求めたり、バグが再現する昔のビルドを使ってバグを再現させたりするなどして、再現手順を確認する必要があります。

## 「再現しない」 (Works For Me)

バグが再現しない場合、開発者は「再現しない」という形でバグ報告票を［解決済み］にします。もちろん、テスターがこれに同意できない場合には、再現のために必要な情報(より詳細な再現手順など)を追記して、バグ報告票を開発者に差し戻します。このとき、バグ報告票の状態は［割り当て済み］に戻ります。

## 「重複」 (Duplicate)

同件ということもあります。同じ内容のバグがすでにほかのバグ報告票にて報告されている場合には、開発者は先に書かれたバグ報告票のID番号だけを重複したバグ報告票に追記し、報告の「重複」として［解決済み］にします。それ以上の情報は記入せず、続きはすべて先に書かれたバグ報告票で追跡します。もちろん、テスターがこれに同意できなければ(「重複」した報告ではないと考えられれば)、その理由を追記し、［割り当て済み］として開発者に差し戻します。

重複したバグ報告票を起票すると、リソースを無駄にしますし、バグ報告票データベースから得られる統計情報を不正確にもするので、避けましょう。バグ報告票を起票する前に、バグ報告票データベースを検索する習慣をつけましょう。

## 「バグではない」 (Not A Bug)

その動作がバグではなく、仕様どおりであると開発者が考える場合には、ソースコードに手を入れることなく、このような形でバグ報告票を［解決済み］とします。また、そのよりどころとなる仕様書などへのリンクをバグ報告票に追記します。もちろん、テスターがこれに同意できなければ、これを［割り当て済み］として開発者に差し戻します。

## 「制限事項」 (Limitation)

ソフトウェアによる制限と、ハードウェアによる制限があります。最終ビルドでも、このバグが制限事項として残る場合には、「制限事項」という形で［解決済み］とします。開発中だから制限事項であって、そのうち制限解除するという場合には、このような形で［解決済み］としてはいけません。制限解除できたビルドにて、「修正済み」という形で［解決済み］とします。

## 「第三者のソフトウェアのバグ」 (Third Party Software Bug)

購入したライブラリやコンポーネント、データベース製品などにバグがある場合です。当該のソースコードがない場合には、開発者は手の出しようがありません。第三者(ライブラリやコンポーネントのベンダ)にバグをレポートして、修正を依頼しなければなりません。この場合、第三者からの返答が来るまで［解決済み］にするのを待つことがあります。

## 「間接的に修正」 (Indirectly Fixed)

実装が追加されたり、ほかのバグの修正などにより、(いつの間にか)間接的に修正されて再現しなくなった場合です。以前のビルドでは再現するが、最新のビルドではなぜだか再現しない場合にこの処理方法を選択します。本当に修正されたのか、どの部分のソースコードの修正により解決されたのかが不明なため、この処理方法はなるべく選択したくないところです。

## 「このバージョンでは修正しない」 (Postpone)

★★次のメジャーバージョンを開発するときには修正するが、現在開発中のバージョンでは諸般の事情により修正しません。これにより［解決済み］とされて、テスターの承認を得ていったんクローズされたバグ報告票は、次のバージョンの開発が始まったときに再開 (Reopen) されなければなりません。同じ製品の次のバージョンの開発を間近に控えているようなときは、そのバグ報告票の状態を再開する (差し戻す) 工数を削減するため、この処理方法により［確認済み］とされたバグ報告票についてはクローズしないこともあります。このときは、このバグ報告票のマイルストーンやバージョン情報をこの次のリリースの番号に変更しておくなどします。

## 「修正しない」 (Won't Fix)

それがバグであることは開発者も認識しているが、バグの深刻度に対して修正のコストが見合わない、あるいはリグレッションリスクが大きすぎるなどの場合には、このような形で［解決済み］とします。テスターが同意できるように、修正しない理由を追記しなければなりません。ユーザーにも納得できるような理由がなければ、テスターも「修正しない」という解決に同意することはできません。また、バグ報告票の状態が［新規］のときに、トリアージチームにより、開発者が着手することなく (つまり［割り当て済み］という状態を経ずに)「修正しない」というアクションがとられてクローズされることもあります。

## 「お蔵入り」 (Obsolete)

UIやアーキテクチャが大きく変更されたことにより、再現手順のとおりに操作できなくなったなど、報告されたバグが古い内容となり、再現手順に意味がなくなった場合はこのような形で［解決済み］とします。これにテスターが同意した場合には、そのバグ報告票の再現手順はテストケースとしても利用されません。つまり、リグレッションテストのインプットとしては使われなくなります。

## 「情報不足」 (Need More Information)

再現手順が曖昧など、バグを再現するための情報が不足している場合には、開発者はいったんこのような形でバグ報告票を［解決済み］とすることがあります。テスターは、より詳細な再現手順を追記して開発者に差し戻します。しかし、新しいバージョンで再現しなくなったなどの場合には、このバグ報告票の状態をそのまま［確認済み］にしても構わないでしょう。

このほか、ソフトウェアのドメインによっては、ここに挙げた以外のバグの解決方法があるでしょう。必要なものを足して、不要なものを削り、皆さんのドメインに最適の処理方法のセットを作成してください。

# バグ報告票に記載すべき項目

バグ報告票の書式に用意すべき項目には、さまざまなものがあります。本書では、一般に使われる項目について紹介しますが、ここに紹介した項目すべてを使うべきというわけではありません。文書の項目が多過ぎると、書くのも読むのも面倒になるし、どこに何を書くべきか分かりにくくなってしまいます。前節で紹介したバグの処理方法と同様、みなさんのプロジェクトで必要な項目のセットを作成してください。

表5 - 3 バグ報告票に記載すべき項目と別名 (例)



## バグ報告票の番号 (Bug#)

ユニークなIDを記入する項目です。ほかの文書(ほかのバグ報告票やメールなど)から、このバグ報告票を参照するときに使われます。同じ書式でたくさん書かれる文書には、どんなものであれ、通し番号を付けるべきです。メーリングリストやメールマガジンなども、よく通しの番号が振られていますね。それと同じです。もっとも、一意にバグ報告票を特定できれば、このIDが必ずしも「通し」の番号である必要はありません。このIDの中に、製品名の略称や、バグが発生したビルドの番号、報告者のイニシャルなどを含めるといった工夫もできます。

## 件名 (Title)

バグを、簡潔かつ具体的に説明する項目です。具体的なバグの内容が想像できる内容にします。例えば、「バグを見つけました」とか、「入力画面のバグ」といったものではいけません。例えば、メールソフトなら「大きなファイルを添付できない」とか、施設予約システムなら「会議室の定員に小数点数を入力できてしまう」といった内容にします。このほか、関連するバグ報告票を簡単に検索できるように、件名の先頭にプロジェクト名やサブシステム名を入れる運用をすることもあります。

## 製品名/ブランチ名 (Product Name/Branch Name)

バグが再現した製品名を記入します。また、ブランチが複数ある場合には、バグが再現したブランチ名も記入します。

## マイルストーン (Milestone)

このバグをどのマイルストーンまでに修正すべきかを記入します。例えば、M1とかM2、あるいはBetaなどと記入しておきます。もちろん、マイルストーンのスケジュールはチーム内で共有されていなければなりません。これにより、このバグの修正をそのマイルストーンの終了条件(Exit Criteria)に含めます。最初に記入したマイルストーンの期間内に修正することが難しければ、このバグの修正を延期するためにこの項目を書き換えます。

## サブシステム名(Subsystem)

バグが認められるサブシステムの名前を記入する項目です。プロジェクトリードは、この情報を参考にして担当者をアサインします。開発するソフトウェアの規模が大きく、また報告されるバグの数が非常に多いとき、このような情報がないと適切な担当者を速やかにアサインすることが難しくなります。サブシステム名を階層(ツリー)構造で管理できるBTSもあります。

## バグの種別 (Issue Type)

このバグ報告票の種別を登録します。例えば、次のようなものを用意しておくと便利です。

表5 - 4 バグの種別の選択肢 (例)

|  |  |
| --- | --- |
| Issue Type | 意味 |
| Code Defect | 製品のバグ |
| Test Issue | テストのバグ |
| Feature Work | 新規機能 |
| Customer Support | ユーザー対応 |
| Work Item | 作業、タスク |
| Document | 文書書き作業 |

## 各担当者名 (Assign to/Developer/Tester)

現在の担当者、開発(修正)担当者、テスト担当者を記入する項目です。バグ報告票の状態が変化したとき、多くのBTSは現在の担当者名を自動的に書き換えます。もちろん、具体的な担当者の名前を記入しておく必要があります。プロジェクトメンバーは、自分の名前が記入されているバグ報告票を追跡することで、これらを自身のタスクリストとして扱えます。

## 状態 (Status) と処理方法 (Resolution)

状態と処理方法については、それぞれ5-2節と5-5節で説明しました。バグ報告票のライフサイクルを管理する上で、特に重要な項目です。

## 再現手順 (Steps to Reproduce)

詳細な再現手順を記入する項目です。バグを再現するための具体的かつ直接的な情報となります。再現するために必要かつ十分なだけの情報を簡潔に書くことが肝要です。バグを再現できる最小のセットを作成することはバグの原因を突き止めるうえで非常に重要なことです。再現手順によっては、テスターがテストケース管理ツールに転記して管理することもあります。

## 観察される事象 (Observed) / 期待される事象 (Expected)

例えば、あるバグ報告票に次のような再現手順が記載されていたとしましょう。

表5 - 14 品質の悪い再現手順

|  |
| --- |
| **Steps to Reproduce:**  1. 会議室予約の画面を開く。  2. 会議室予約のボタンを押下する。  3. 「会議室が予約されました」というメッセージが表示される。 |

この再現手順には問題があります。どこでしょうか。

答えは、最後の3.のステップです。これでは「会議室が予約されました」と表示されるべきなのに表示されないのか、表示されないべきなのに表示されるのかが分かりません。

これでは、開発者がこの手順を試しても、バグを再現できているかも分かりませんし、どう修正すべきなのかもわかりません。これを避けるには、観察される事象と期待される事象を、再現手順とは区別して記入します。期待される事象には、その根拠となる仕様書へのリンクを記載すれば、もっと素敵です。BTSには、この項目を独立して用意していないものも多いですが、そのような場合は、再現手順の項目に、この項目を合わせて記載しましょう。例えば、次のようにします。

表5 - 15 品質の良い再現手順

|  |
| --- |
| **Steps to Reproduce:**  1. 会議室予約の画面を開く。  2. 会議室予約のボタンを押下する。  **Observed:**  「会議室が予約されました」というメッセージが表示される。  **Expected:**  「予約したい会議室を選択してください」というメッセージが表示される。  **Note:**  この仕様の詳細については、機能仕様書の3-8節 会議室予約をする を参照してください。 |

## 添付資料 (Attachments)

バグの調査に都合のよい資料があれば添付します。再現時のログファイルや、再現手順を補足説明する画面ショット、再現のためのテストデータなどです。再現時のログや画面ショットは、確かにバグが再現したという証拠になりますから、バグの再現性が低いときにはぜひ添付したいところです。それがあれば、再現性が悪いときでも、開発者は簡単に「再現しない」という形でバグ報告票を解決できません。画面ショットを保存するとき、.bmp形式ではファイルサイズが非常に大きいのでディスク容量を圧迫します。.pngや.gifなどの圧縮された形式を使いましょう。

## 再現性 (Reproducibility)

再現頻度のことです。“いつも再現する” とか、”ときどき再現する” などと記入します。あらかじめいくつかの選択肢を用意しておくと良いでしょう。あるいは、再現手順を試行した回数を分母に、再現した回数を分子にして、2/5などと分数で表記するのも良い習慣です。私はここに2/500と書かれた報告票に、テスターの真摯さ(というか根性)を見たことがあります。このような再現性の低いバグは修正が困難ですが、500人のユーザーが毎日使えば、1日に2人はこのバグを見ることになります。このような側面からは、2/500は再現性として低くはありません。トリアージの際は、考慮が必要です。

バグを修正するには、まずそのバグを再現させることが必要となることがほとんどです。つまり、再現性が高い(良い)バグは修正が容易ですが、再現性が低い(悪い)バグは修正が困難なのです。再現性が悪いバグを見つけたら、より再現性が良い手順や環境を探してみましょう。

## 深刻度と優先度 (Severity / Priority)

深刻度(シビリティ)は、このバグがどれだけ深刻か(シビアか)を示すもので、5～10段階の数値がよく使われます。Severityは重要度と訳されることもありますが、深刻度もしくは重大度とした方が適切と思います[[4]](#footnote-4)。バグとは、ソフトウェアの「病気」もしくは「けが」のようなものなのだからです(深刻なけがのことを、重要なけがとはいいませんね)。

優先度(プライオリティ)は、このバグを解決する優先度を決める順位で、3～5段階の数字がよく使われます。次にどのバグの解決に着手すべきかを開発者が判断できるように、優先度でバグ報告票を順序付けます。

深刻度と優先度の重要な違いは、深刻度は絶対基準であり、優先度は相対基準であることです。表5 - 5に、深刻度の基準の一例を示します。この基準に従えば、誰がつけても深刻度は同じ数字になることがわかるでしょう。また、このような基準に従えば、深刻度と再現性を区別することも容易です。再現頻度が多いから高い深刻度をつける、などということをせずに済みます。

表5 - 5 深刻度の基準 (例)

|  |  |
| --- | --- |
| 深刻度のレベル | 意味 |
| 0=Enterprise wide crash | 再現時には、複数のPCが同時にクラッシュ |
| 1=Crash or system hang | このPCがクラッシュもしくはハングアップ |
| 2=Data loss/feature failure | ユーザデータを失う、もしくは機能が失敗する |
| 3=Not working as expected | 期待通りに動かない |
| 4=Performance | 処理速度が遅い |
| 5=User friendliness/usability | 使いにくい |
| 6=Cosmetic | 見た目がおかしい、誤字や脱字があるなど |
| 7=Enhancement | 治ったらうれしい |
| 8=Documentation | ドキュメント (ヘルプ) の誤り |

これに対して、優先度というのはあくまで相対的な基準です。仮に、すべてのオープンなバグ報告票の深刻度が同じく1だとしても、その優先度も全部1に設定したら、優先度をつけたことになりません。それでは、開発者はどのバグから着手したらいいのか分かりません。

とはいえ、何がしかの基準がないと優先度も付けにくいものです。そこで、バグを修正するまでの期限を優先度の基準として用意することもあります。この一例を表5 - 6に示します。もちろん、このような基準を適用する場合でも、優先度をつけるときはほかのバグの数や状況に応じて調整が必要なことは変わりません。

表5 - 6 優先度の基準 (例)

|  |  |
| --- | --- |
| 優先度のレベル | 意味 |
| P1 | 最優先で修正 |
| P2 | このマイルストーンが終わるまでに修正 |
| P3 | 製品をユーザーにリリースするまでに修正 |
| P4 | 次のバージョンで修正 |

このほか、緊急度 (Urgency) という項目が使われることもあります。緊急度には、0=すぐに対応, 1=24時間以内に対応、2=3営業日以内に対応、といった基準を設けることが多いようです。私の主観では、ソフトウェア開発のためのバグ報告票には緊急度という項目は使いにくいと思います。というのは、バグの修正に必要な時間を見積もるのは難しいですし、すぐに治せと言われても無理だからです。緊急度は、ユーザー対応やソフトウェア保守のためのチケットシステムではよく使われますが、これはユーザーに最初の回答を用意するまでの期限という意味であることが多いようです。

表5 - 7 深刻度と優先度の違い

|  |  |  |
| --- | --- | --- |
|  | Severity | Priority |
| 基準 | 絶対的 | 相対的 |
| 観点 | 客観的 | 主観的 |
| 複雑さ | 単純 | 複雑 |
| 視点 | 単視眼的 | 複視眼的 |
| 意味あい | 技術的 | ビジネス的 |
| 目線 | ユーザー目線 | ビジネス目線 |

# コラム 絶対基準の項目には、その基準を表示すべし

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **コラム 絶対基準の項目には、その基準を表示すべし**  私は、毎月アマゾンでたくさんの本を購入します。アマゾンのサービスにはおおむね満足していますが、とても残念に思うことがあります。それは、レビューアが商品を★で採点するとき、その絶対基準が画面に表示されないことです。このため、商品を購入するとき、この★の数はほとんど参考になりません。特に、レビューアの数が少ないときは★の数を参考にしない方が賢明です。ユーザーが書籍を評価するときには、たとえば次のような基準を画面に表示して欲しかったと思います。  5=今年のマイベストの1冊  4=そこそこ面白い。買って損はない  3=人を選ぶ  2=時間とお金の無駄。買わない方が良い  1=世間に害悪をなす本。焚書にすべし  このような、評価点数を登録できるサービスは、ほかにもDVDレンタルなどのサイトでよく見かけます。が、点数の基準が示されているものはほとんど見たことがありません。  これに対し、私が愛用している「みんなのシネマレビュー」という映画評価サイトは、適切な基準を設けています。このサイトでは、レビューアが10点満点で映画を評価でき、映画ごとにレビューアの数が点数別に集計されてグラフに表示されます。このサイトが素晴らしいのは、レビューを投稿するとき、画面に点数の基準がきちんと表示されることです。この基準を表5 - 10に引用します。もちろん、このような基準を設けても、映画の評価は人によって違うものです。例えば、ある人には10点満点の映画でも、別の人には同じ映画が0点ということがあります。しかし、このレビューサイトで興味深いのは、レビューアの数が十分に大きくなると、多くの映画で点数の分布が正規分布に近くなることです。「ダンサーインザダーク」のような、人によって極端に評価が分かれるような映画であっても、点数の分布がなだらかに連続しています(とびとびの値に偏りません)。それは、このサイトで表示される点数の基準が良くできていることの証拠でしょう。みなさんが BTS を運用するときも、バグをファイルするときに各項目に適切な基準が表示されるように工夫してください。そうすれば、十分に多い数のバグ報告票がファイルされたとき、意味のある数字がバグ報告票データベースから収集できるようになります。  表5 - 10 みんなのシネマレビューで使われている映画の評価基準   |  |  | | --- | --- | | **点数** | **内容** | | 10 | 満点の評価  傑作中の傑作。ここ何年間で最高の作品。 | | 9 | 傑作とも言える作品。  1年で一番の映画。 | | 8 | 見た後、率直に面白かったぁ・・  って言える作品。  出来としては良い。 | | 7 | なかなか面白い作品。  他人に薦められなくもない。 | | 6 | そこそこに面白い作品。  ま、損はしてないかなって感じ。 | | 5 | 可も不可もなくってとこ。  損も得もしてないっすね。  ビデオでも良い感じ。 | | 4 | 意外と面白くない。  全然期待しないで行くとひょっとしたらひょっとする。 | | 3 | かなり面白くない。  かなりコケ具合。  そのコケ具合を誰かに言いたい。  別名。しょぼ映画 | | 2 | 年に一度あるかないかのはずれ映画。  いろんな人に警告したい映画。  ある意味貴重 | | 1 | メガトン級の面白なさ。  面白無さ過ぎて大笑いしそうな映画。  ここ何年かで一番面白くない映画。 | | 0 | 長い映画人生で一番面白くない映画。  っていうか、映画っていいたくない作品。  超激しょぼ映画  でも、ある意味貴重 |   © <http://www.jtnews.jp/> |

## トリアージの判定結果 (Triage)

トリアージとは、このバグを修正すべきかせざるべきかを判断することです。この項目に用意すべき選択肢としては、表5 - 11のようなものが適当です。トリアージについては、後の節で改めて扱います。

表5 - 11 トリアージの判定結果の選択肢 (例)

|  |  |
| --- | --- |
| トリアージの判定結果 | 意味 |
| Triage Needed | 要トリアージ (未トリアージ) |
| Fix | 修正する |
| Don’t Fix | 修正しない |
| Recommend Fix | 修正を推奨 |
| Recommend Don’t Fix | 修正しないことを推奨 |

## どのように発見したか (How Found)

どのような手段でバグを発見したかを記録しておきます。例えば、選択肢としては次のようなものが適当です。

表5 - 10 どのように発見したかの選択肢 (例)

|  |  |
| --- | --- |
| どのように発見したか | 意味 |
| Automation | 自動化したテストで |
| Buddy Build | バディビルドで |
| Bug Bash | バグバッシュで |
| Build Break | ビルドブレークで |
| BVT | Build Verification Testで |
| Code Review | コードレビューで |
| Development | 開発中で |
| Dogfood | ドッグフードで |
| Unit Test | 単体テストで |
| Spec Review | 仕様レビューで |
| Test Pass | テストケースによるテストで |
| Ad Hoc | その他 |

## 環境 (Environment)

バグが再現した環境を記入する項目です。バグが見つかったソフトウェアを動作させていたマシンのCPUやメモリ容量、OSのバージョンなどを記入します。このほか、ソフトウェアのドメインによって、この項目に記述する内容は変わってきます。例えば、Webアプリケーションであれば、ブラウザの種類とバージョンもこの項目に含まれるべきでしょう。バグを再現するための非常に重要な情報となりますから、みなさんが開発するソフトウェアのドメインに応じて、この項目を過不足なく準備してください。また、バグが再現しなかった環境があれば、このバグ報告票のほかの項目(再現手順やコメントなど)に記録しておくと良いでしょう。

## 修正時のチェンジセット番号 (Changeset#)

開発者は、このバグの修正をリポジトリにコミットするとき、このバグ報告票の番号をSCMのコミットコメントに記入します[[5]](#footnote-5)。また、そのチェンジセット番号をバグ報告票に記入します。SCMのチェンジセット番号とBTSのバグ報告票を双方向に関連づけることで、このバグの解決がどのソースファイルの修正によるのか、またソースファイルの修正がどのバグを修正したのか、後で追跡できるようになります。このような追跡可能性をトレーサビリティ (Traceability) といいます。追跡可能性を確保することは、バグ追跡システムにとって重要です。

## バグの再現を確認したバージョン/ビルド番号 (Reproduced Version / Build#)

テスターがバグ報告票を新規に作成するとき、そのバグを再現した (発見した) ビルドの番号を記入する項目です。一般的な BTS には、これをひとつしか記入できないものが多いですが、複数のビルドでテストした場合には、バグ報告票の再現手順などの項目に、次のようにテスト結果を漏らさず記入しましょう。

|  |
| --- |
| **Test result:**  build#35 OK  build#36 OK  build#37 NG  build#38 NG  build#39 NG |

この場合には、build#37で入ったバグであることがわかります。このときは、もしこのバグを最初に発見したときのビルド番号が39であったとしても、この項目にはバグが再現した一番若いビルドの番号 (上記の例では37) を記入するようにします。バグが再現する一番若いビルドの番号を素早く探す方法については、★3-13節を参照してください。

## バグを解決したバージョン/ビルド番号 (Resolved Version / Build#)

開発者が、バグを解決 (修正) したビルド番号を記入する項目です。実際にバグを修正する前に、この項目を埋めても良い(あるいは埋めるべき)という運用もあります。その場合、バグ報告票の状態が［割り当て済み］のとき (このバグが修正される前) は、この項目は見積もりを表します。また、状態が［解決済み］のとき (バグが修正された後) は実績を表します。

このバグを修正したときは、この項目(実績)の記入は必須です。ここに書かれた番号のビルドがリリースされたら、テスターはその修正を確認するテストを開始するからです。

## バグの解決 (修正) を確認したビルド番号 (Verified Build#)

このバグが「修正済み」という形で解決された場合、テスターはその修正を確認し、このバグ報告票を［確認済み］とします。その際、確認に使ったビルドの番号をバグ報告票に記入します。つまり、このバグ報告票の再現手順 (テスト手順) にパスしたビルド番号を記入する項目となります。この項目をバグ報告票の書式に用意せず、コメントの項目などに「build#23で修正を確認しました」などと記入しても良いでしょう。

## 回避方法(Workaround)

このバグを回避する運用方法があれば記入します。開発者がバグの原因を突き止めるためにも、価値の高い情報となります。また、テストチームがテストする際のインプットにもなります。この回避方法を、最終ビルドのリリースノートに転記する形で、ユーザーに伝えることになるかもしれません。もし回避方法を発見したなら、忘れず記入してください。

## 履歴とコメント (History / Comment)

時の担当者は、このバグ報告票の状態を遷移させるときに、その根拠を追記します。また、このときの日時も合わせて追記します。バグ報告票が更新・保存されたとき、そのときの日時や、その保存前後の項目値の差分を自動的にこの項目に記録するBTSもあります。

これらすべての項目が原因の調査と修正の作業に役立つわけではありませんが、バグ報告票データベースを集計することで、有意な統計情報を収集できる項目があります。また、すべての項目を起票時に埋めることはできません。各担当者間でやりとりする間に、必要な情報が集まってきます。



図5 - 4 バグ報告票を起票するステップ  
(4番目のステップについては、★3-14節を参照)

# バグ報告票によるフィードバック

★図2-5を再掲します。この図のように、テストチームによるテストと開発チームによる開発は、同時並行で行われます。元の図に、ななめの矢印を加えました。この矢印は、テストチームが書いたバグ報告票が、開発チームに受け渡されるようすを示しています。



図5 - 5 発見されたバグが、次のビルド開発に  
フィードバックされるようす

つまり、前のビルドで発見したバグは、次のビルドの開発にフィードバックされ、開発を正しく方向づける一役を担います。段取りの良いソフトウェア開発のためには、バグ報告票の品質がとても重要です。ウィークリービルドのアプローチを使うときは、このサイクルが一週間を単位として回ることになります。



図5 - 6 ビルドのリリースとバグの報告のサイクル

# バグ追跡のアンチ・プラクティス

バグ報告票の管理と運用は、知らない人には結構難しいものらしく、間違ったやり方をよく見ることがあります。私がこれまでに見たことのある間違いを紹介します。

## 状態と処理方法を一緒くたにしている

私がこれまでに最も多く目にした間違いは、バグの［状態］と「処理方法」を混同してしまうことです。特に、BTSを使ったことのない人がスプレッドシートでバグを追跡するとき、この間違いを犯す傾向があるようです。例えば､バグ報告票の取り得る状態として［修正済み］や［再現しない］を用意してしまうと、現在そのバグがどのような状態にあるのか、分からなくなってしまいます。状態と処理方法は明確に区別しましょう。

## バグ報告票の状態が多すぎる

前節と関連しますが、バグ追跡のワークフローを適切にカスタマイズしようとして、逆に不要な状態を多く用意してしまうことがあります。バグ報告票が取ることのできる状態の数はそう多くありません。本章で紹介した以外の［状態］を追加するときは、慎重に行ってください。状態はそのままで一時的に別の人をアサインし、バグ報告票に情報を追加して返してもらうこともできます。ワークフローはシンプルな方が、バグ報告票をスムーズに流せます。

## バグ報告票の項目が多すぎる

初めてBTSを導入しようとするときは、バグ報告票の項目数は必要最小限に抑えましょう。項目数が多いと、読むのも書くのも面倒になって、導入に失敗してしまいます。オプショナルな項目は、すべてコメント欄にまとめて書くのも良い方法です。また、BTSには新規にバグを起票するときのテンプレートを用意できるものがあります。このような機能をうまく使って、バグを起票するときの工数を減らしましょう。

## バグ報告票の妥当性を検査するプロセスがない

バグ報告票を起票できる権限が複数人に与えられている場合、相矛盾する内容のバグが報告されてしまうことがあります。このため、バグ報告票が記述されたら、まずその妥当性を検査して、開発者の解決作業にゴーサインを出すプロセスが必要です。不適切な報告票に基づき、開発者が作業しても仕方がありません。このようなモデルをV&V (Validation & Verification) といいます。Validation (妥当性検査) とは、「正しいソフトウェアを作っているか」、Verification (確認・検証) とは「正しくソフトウェアを作っているか」ということです。先に示したバグ報告票の［Open］という状態は、［Validated］と呼ぶべきかもしれません。この問題に対する単純な解決のひとつは、バグをオープンできる権限を少人数に絞ることです。複数人がバグを報告してもよいが、それらをオープンできるのはテストリードだけ、といった形にします。あるいは、テストリードやデブリードでトリアージチームを編成し、そのメンバーだけがバグ報告票をオープンできるようにしても良いでしょう。

## バグ報告票の解決を確認するプロセスがない

開発者がバグを修正したとき、バグ報告票を［割り当て済み］から［解決済み］を経ずにいきなり［確認済み］あるいは［終了］にしてしまいます。前節のValidationのプロセスがないという問題に対して、こちらはVerificationのプロセスがないという問題です。開発者が修正したと報告したバグは、本当に修正されたのか、テスターがビルドをテストして確認(verify)する必要があります。

## ビルド番号が管理されていない

単に「いついつの時点でこのテストケースを実施したら結果がOKだった」といわれてもすごく困ります。重要なのは、いつテストしたのかではなく、どのビルドをテストしたかです。どのビルドでバグを発見したのか、どのビルドでバグを修正したのか、正しく追跡できるように、ビルドには必ずビルド番号を付加しましょう。

## 開発者とテスター間のコミュニケーションが密でない

効率よくバグを解決していくには、開発者とテスターとの間で密に段取りを付ける必要があります。根回しをおろそかにすると、バグ報告票がたらい回しになり、いつまでたっても解決できません。次のようなやりとりを考えてみましょう。



図5 - 7 コミュニケーションが密でないと…

これでは時間を無駄にするばかりか、人間関係が崩壊してしまうかもしれません。これを避けるには、必要な情報を漏れなく丁寧にやりとりします。必要に応じて、バグ報告票だけでなく、メールやF2Fなどの手段も使ってコミュニケーションしましょう。チャットも便利です。

## 深刻度と優先度を一緒くたにしている

バグ報告票は、各担当者のタスクリストのような形で処理しますから、どんな組織でもバグ報告票に優先度(らしきもの)を付けることはしているようです。しかし、深刻度と優先度はまったく別の概念ですから、優先度 (らしきもの) だけでバグ報告票を処理しようとすると、人や開発のフェイズに依存してその重み付けの基準が変わってしまい、バグ報告票を正しく順序付けることができなくなります。それぞれ別の側面に注目し、適切な尺度を与えることができるように、深刻度と優先度は独立して準備しましょう。

## テスターのモチベーション管理に失敗する

テストチームのモチベーションを維持するのは難しいものです。「どんなに頑張っても、自分は開発中のソフトウェアの品質向上には貢献できないのではないか」といった無力感を感じると、テスターはやる気を失います。テスターが報告したバグに対して、開発者が適切なタイミングで適切なアクションを取ってあげなければ、テスターはバグ報告票を記述する作業に価値を見つけることができません。せっかく多くのバグを報告しても、これらに対して何のアクションも取られないまま、それらの再現手順が古いものになってしまっていくと、テスターは転職を考え始めます。

また、ソフトウェア開発をよく理解していない組織では、バグを多く報告するテスターを不当に低く評価することがあります。バグが多く見つかるほど、それらを修正するコストが多く掛かりますから、バグは見つからない方がいいというわけです。しかし、これは明らかに誤りです。テストは、バグを発見するために実施するのですから、むしろ1週間に発見するべきバグ数のノルマを決めて、各テスターのモチベーションをケアすべきです。毎週、バグを一番多く発見したテスターをMVPとして表彰する、といった工夫も有効です。

テスターは、仕様の詳細化について強い権限を持つ係です。ユーザーのドメインに明るく、開発中のソフトウェアの仕様について責任をもつ人がテストを担当します。ユーザーの要求を獲得するプロセスにも、レビューなどに参加するなどで、テスターがかかわっていくべきです。残念なことに、日本では「テスター」に強い権限を委譲することは少ないため、プロ意識を持ったテスターが育つ土壌が乏しいのが現状です。テストと開発プロセスには密接な関係がありますから、テストチームは (も) 開発プロセスを改善する責務をも担うべきです。いかにテスターという仕事が生産的で創造的なものか、理解していただけるでしょうか。もちろん、網羅性のあるテストケースの設計と実装、自動化という困難な仕事も、テスター (テストエンジニア) の責務です。

## バグの数を、社員の成績評価に使う

開発者を修正したバグの数で、テスターを発見したバグの数で、評価して昇給やボーナスを査定する組織があります。しかし、このような取り組みはたいていうまくいきません。バグによって修正の難易度がまったく違いますから、修正したバグの数で開発者を正しく評価することはできません。また、このような基準を持ち込むと、テスターが些細なバグばっかりを大量に報告することがあります。修正が容易なバグがたくさん報告されると開発者も嬉しいので、プロジェクトチームを誤った方へ方向づけることになります。バグをたくさん修正した開発者や、バグをたくさん発見したテスターをみんなで称えるのは、仕事を楽しくし、やる気を向上する効果があります。しかし、これをそのまま給与の査定に使うのは仕事をつまらなくしてしまうばかりか、バグ報告票の品質にも悪い影響を及ぼします。

# リグレッション

ある修正を入れたことで、以前動作していた機能が動かなくなる不具合をリグレッションバグ、あるいは単にリグレッションといいます。Regressionには退化とか後退、後戻りなどの意味があります。ソフトウェアを進化させていくソフトウェア開発において、regression (退化) という言葉は忌むべき現象をよく表しています。また、このようなバグを指してデグレードということもあります。デグレードとは、文字通り品質のグレード (等級) が落ちる、という意味です。どちらも同じ意味ですが、パフォーマンスなどの非機能的な側面が以前より悪くなるときは、デグレードという言葉を使うことが多いようです。



図5 - 8 リグレッションバグ

動いているソフトウェアを修正すると、リグレッションが発生する危険があります。特にソフトウェア開発の末期では、リグレッションを恐れるべきです。予定している最終ビルド日が迫っている中でリグレッションが発生したら、(それを発見できるべきことは当然としても) 修正する時間は残っていないかもしれません。

動いているが汚いコードに手を入れて、きれいなコードに書き直し、今後の開発や保守を楽ちんにしよう、というのがリファクタリングという習慣です。しかし、リファクタリングでも、(アーキテクチャやコード記述の美しさではなく) 十分なカバレッジをもつ自動化されたテストによってソフトウェアの振る舞いが変わらないことを担保し、リグレッションを回避すべし、とされています。つまり、コードに手を入れたら、必ずテストをしてリグレッションを漏れなく発見しなければなりません。リグレッションを発見すべきことは当然としても、それを限られた時間で修正できるとは限りません。ソフトウェア開発の末期段階(リリース直前)では、リグレッションを恐れ、コードを修正することに臆病になるのがエンジニアの正しい態度です。

# リグレッションテスト

前節で述べたように、コードを少しでも修正したら、リグレッションが発生する危険があります。リグレッションを発見するには、過去のビルドでパスしたテストケースをもう一度やり直すことが有効です。このように、すでに実施済みのテストケースをやり直すテストをリグレッションテストといいます。このテストケースには、いままでに記述されたすべてのバグ報告票にあるバグの再現手順を含みます。

Regressionという語には、先に紹介した退化や後退といった意味のほかにも「回帰」という意味があり、リグレッションテストは回帰テストと訳されます[[6]](#footnote-6)。回帰とは、1周してもとへ帰るとか、繰り返すということです。つまり、リグレッションテストという名前には、リグレッションを発見することと、何度も繰り返して行うことの両方の意味が含まれています。このテストの性質を非常に良く表しているといえるでしょう。

余談ですが、回帰といえば鮭を思い出します。鮭はもともと白身の魚ですが、生まれた川に回帰するときに身が赤くなるのだそうです。しかし、ソフトウェアテストの世界では、いままで通っていたテストをリグレッションしてみたらテスト結果が真っ赤に染まっていた、なんてことにならないようにしたいものです(多くのテストツールは、テストの成功を緑で、失敗を赤で表示します)[[7]](#footnote-7)。

これまでに書かれたすべてのバグ報告票とテストケースによるリグレッションテストを、フルリグレッションテストといいます。ユーザーにリリースする直前にはフルリグレッションテストを実施して、治したはずのバグが再発していないことを確認しなければなりません。リグレッションテストの実施には大変な工数がかかります。開発中のソフトウェアの規模にもよりますが、筆者の経験ではテストチームのメンバーを総動員して3日間から5日間ほどのイベントになります。リグレッションテストの実施には、いくつか工夫が必要です。

## テストを自動化する

一番望ましいのは、テストを自動化しておくことです。そうすれば、すべてのビルド (例えばデイリービルド) に対してリグレッションテストを実施でき、発生したリグレッションをすぐに発見できます。最近のコード修正によるバグなら、修正も比較的簡単です。困るのは、テストを自動化することにもコストがかかることと、そもそも自動化するのが不可能なテストケースもあることです。テストの自動化にどれだけ投資すべきか、よく検討しましょう。また、多くのテストを自動化した場合でも、自動化できないテストのために、やはりマニュアル(手作業)によるリグレッションテストのイベントも企画しましょう。

## リグレッションテストを実施する頻度を減らす

マニュアルケースのリグレッションテストを毎日実施することは、とても現実的ではありません。各マイルストーンの最後に1回ずつ実施するなどのようにします。

## すべてのテストケースを実施しない

リグレッションテストを実施する工数を減らす一番簡単な方法は、全部のテストケースを実施しないことです。修正したソースコードの影響範囲からリグレッションすべきテストケースの範囲を絞り込むとか、機能ごとにリグレッションテストのセットを作成しておくなどの方法があります。また、重要なシナリオを含むバグ報告表の再現手順に限ってテストケースに転記しておき、リグレッションテスト時には、バグ報告票データベースをインプットとしては使わないという方法もあります。ただし、実施するテストケースの範囲は、慎重に検討してください。特に、ユーザーにリリースする直前のリグレッションテストは、安易に省くべきではありません。

# トリアージ

いよいよ開発中のソフトウェアが完成に近づき、最終ビルド候補のフルリグレッションテストが終わったと考えてください。この結果、不幸にもバグを発見したときは、利害関係者を招集してトリアージミーティングを行い、発見したバグを治すべきか、それともそのままリリースすべきかを判断します。トリアージとはもともと野戦病院で使われていた言葉で、傷病兵の負傷の程度による治療の優先順位を指します。転じて、危機的状況における優先順位の決定原理を意味しています。

野戦病院では、決定的にリソースが足りません。医者、看護婦、ベッド、薬、包帯…… 、何もありません。可能な限り多くの兵隊さんを助けるため、治療の対象とする兵隊さんを選別する必要が出てきます。治療の優先度に応じて、兵隊さんに黄や赤のタグ (トリアージタグ) を付けていきます。ひん死の兵隊さんは治療の対象になりません。治療するためのリソースは限られているのですから、どんなに手を尽くしても助けられる見込みがないような深刻度の高いけが人は、優先度が一番低いことを示す黒いタグを付けます。ひん死の人を見捨てるという、つらい判断をしなければならないのです。逆に、死なない程度に重傷の人であれば、やはり優先度も低くなりますが、それがたった1人しかいない通信兵なら、深刻度は低くても、治療の優先度が高くなることもあるでしょう。

ソフトウェア開発の現場も戦場です。決定的にリソースが足りません。時間、人、マシン、ライセンス、お金……、何もありません。可能な限り最終ビルドに含まれるバグを減らすため、修正の対象とするバグを選別する必要が出てきます。深刻なバグだからといって、修正の対象になるとは限りません。修正するためのリソースは限られているのですから、修正に大きなコストが掛かるようなバグは優先度としては低くなります。深刻なバグを開発の末期段階まで残してしまうと、ひん死の人を見捨てるのと同様の、つらい判断をしなければならないのです。逆に、許容できる程度に深刻度が低いバグであれば、やはり優先度も低くなりますが、深刻度とは違う視点から (マーケティング上の理由などにより)、修正の優先度が高くなることもあるでしょう。

ただし、トリアージという語を使うときには、優先順位を付けるよりも「治すか治さざるべきか」の判断が強調されることに注意しましょう。優先順位をつけることはプリオリタイズ(prioritize)といい、トリアージとは明確に区別されます。

このような判断は、最終ビルド候補以外のビルドでも常に行うべきですが、特に最終ビルド候補ではこの観点が重要となります。フルリグレッションテストの後にコードを修正したら、もう一度フルリグレッションテストをやり直さなければならないからです。

バグを修正することで、別のリグレッションバグが発生するかもしれないことをリグレッションリスクといいます。リグレッションリスクは、コード変更の規模や影響範囲、変更前後のコードの複雑性などから開発者が見積もります。リグレッションリスクに対処するには、ユーザーにリリースする前にリグレッションテストを実施するのが有効ですが、フルリグレッションテストの後のコード修正は、テスト工数へのインパクトがとても大きくなります。

ですから、トリアージミーティングでは、修正しないとリリースできないバグ (ショーストッパー) があるか (どれか) を、利害関係者間で調整・判断することになります。修正の対象としなかったバグは、制限事項として最終ビルドのリリースノートに記述します。



図5 - 9 医療で使われるトリアージタグの例 (Wikipediaより)

# トリアージ基準の変化

開発の末期段階では、リグレッションが発生したときの影響が大きくなるため、コードを修正しにくくなります。このため、トリアージの基準も時間とともに変わります。図5 - 10を見てください。これは、X軸に開発時期を、Y軸に修正したいバグの深刻度をとったものです。



図5 - 10 トリアージ基準の変化 (その1)

開発の初期段階では、あまりトリアージに時間をかける必要はありません。まだ動いているものがない (あるいは少ない) のだから、リグレッションリスクを考慮する必要もない (あるいは少ない) し、もしリグレッションが発生しても、それを発見して治す時間がたっぷり残っているからです。

しかし、ユーザーにリリースする直前では、リグレッションリスクやテスト工数へのインパクトを考慮し、回避方法があるバグや深刻度が低いバグは修正をあきらめることになります。また、修正しないとリリースできないような深刻度の高いバグ (ショーストッパー) は、無理をしてでも治すしかありません。が、そもそも、開発の末期まで深刻なバグを発見できなかったり、開発の末期に深刻なバグを埋め込んでしまうことの方が間違っているといえます。

といっても、開発の末期段階では、些細なバグはまったく治してはいけないというわけではありません。図5 - 11を見てください。これは、X軸は図5 - 10と同じく開発時期ですが、Y軸には当該の修正に対するリグレッションリスクやテストインパクトの大きさをとったものです。



図5 - 11 トリアージ基準の変化 (その2)

このように、リグレッションリスクやテストへのインパクトが小さな修正は、開発の末期でもコミットできます。例えば、画面上の誤字が開発の末期段階で見つかったとします。これを治すにはプログラムのロジックを修正する必要はなく、とても安全な修正で済みます。フルリグレッションテストをやり直す必要はないので、テストへのインパクトも少なくて済みます。ただし、このバージョンの製品を修正する機会はもうありませんから、いつもより慎重に作業する必要があります。このようなときは、トリビアル[[8]](#footnote-8)な修正であっても、開発チーム全員でレビューしてからコミットするなどのようにします。

このように、トリアージの基準は時間とともに変化します。修正してよいバグのバーが段々上がっていくということです。3-13節で紹介した、コミットポリシーの段階的な追加も参照してください。バグをトリアージするプロセスは、開発が末期段階になってから導入すると良いでしょう。特に、リリース前のリグレッションテストの実施前後で、トリアージの基準が大きく変わることに注意してください。

最後にひとつ念を押しておきますが、「もしユーザーにリリースする直前に深刻なバグが見つかったら、それは修正しなくても良い」と考えるのは大きな間違いです。リリース直前に深刻なバグが見つかったとすれば、それはそもそも開発プロセスのどこかが間違っているのです。

# トリアージのまとめ

トリアージの語源は、フランス語の「選別」という言葉です。医療でのトリアージは、第一次世界大戦の戦場で、フランス人の医師が始めたそうです。もともとは、フランスで羊毛やコーヒー豆を選別するときに使っていた言葉で、良いものだけを選り抜き、痛みがひどいものは捨てる、というニュアンスがあります。羊毛をトリアージするときには、本書で示したのとはまったく別の基準が必要になることは言うまでもありません。

医療のトリアージが一般に認知される以前は、尊い人命を野菜のように選別し、状態の悪い患者を見捨てるとはけしからん、というとんちんかんな批判があったようです。もちろん、医療におけるトリアージは、限られたリソースを効率的に使ってより多くの人の命を助けるために有効です。

ソフトウェア開発のトリアージは、まだ一般には認知されていないのかもしれません。今でも、深刻な医療現場を想起させるトリアージという語は、ソフトウェア開発では使うべきではない、という批判を聞くことがあります。しかし、ソフトウェア開発におけるトリアージも、限られたリソースを効率的に使って製品の品質を向上させるために、非常に有効な考え方です。

BTSに登録された新機能の提案[[9]](#footnote-9) をトリアージするときにも、本章で示したのとは別の基準が必要です。文脈と状況に応じて正しい基準を使い分け、適切なトリアージをしましょう。

ソフトウェア開発におけるトリアージが医療のトリアージと全く異なるのは、たとえあるバージョンで修正しないとトリアージしたバグであっても、次のバージョンでは修正できるかもしれないことです。図5 - 12に、果物を重量で選別する選別機の写真を使ってトリアージのイメージを示しました。トリアージ済みのバグには、「修正する」ことに決めたバグ以外にも、「修正しない」ことに決めたバグが含まれることに注意しましょう。



図5 - 12 バグの選別

# バグ収束曲線

ここから、BTSから収集できる数字とその使い方についていくつか紹介しましょう。横軸に時間を、縦軸にバグの発生件数をとって折れ線グラフを書くと、一般に図5 - 17のような形になることが知られています。これはバグ収束曲線[[10]](#footnote-10)として知られています。また、信頼度成長曲線 (Software Reliability Growth Curve) ともよばれます。



図5 - 17 バグ収束曲線の例

ここでは、横軸にウィークリービルドの番号を使いましたが、デイリービルドの番号や日付をとっても構いません。この傾きを見れば、リリース可能な品質に達するまでのおおよその時間を見積もれます。気を付けたいのは、バグが収束したように見えても、実は新しいバグを発見できていないだけ、という可能性があることです。必要なテストケースを発見できていないとか、必要なテストケースを実施していないときに、そのようなことが起こります。各ビルドでバグを発見できず、バグが収束してきたようなら、各ビルドでどのテストケースが実施されているか確認したり、バグバッシュやリグレッションテストを企画するなどして、確かにバグが収束していることを確認しましょう。

図5 - 17の例では、だいたいビルド#7を過ぎたあたりで収束に向かっています。このように、バグが収束に向かうには、週あたりでテストチームが発見するバグの数(A)と、同じ期間で開発チームが解決 (修正) できるバグの数(B)の差が重要です。(B)が(A)に追いつけば、たとえ現在オープンなバグの絶対数が多くても、それは減っていきますから、リリース可能な品質に近づく時期を見積もれるようになります。



図5 - 18 バグが収束するまでに必要な時間の見積もり

# ゼロ・バグ・バウンス

図5 - 15を見てください。これは、横軸に時間を、縦軸にオープンなバグの件数をとって、折れ線グラフを書いたものです。



図5 - 15 オープンなバグの件数の推移の例

このように、開発の段階が末期になると、オープンなバグの数は減っていきます。グラフの右の方に注目してください。瞬間的にオープンなバグの数がゼロになるときがあります。これは、開発チームのバグ修正のスピードがテストチームのバグ発見のスピードに追いつき、オープンなバグをすべて解決したことを意味します。グラフがボールが弾むような形になることから、これをゼロ・バグ・バウンス (Zero Bug Bounce) といいます。別のバグがすぐに発見されるので、バグがゼロの状態を維持することはなかなかできません。そこで、ZBBを「48時間以上、オープンなバグがない状態」のように定義することもあります。また、［解決済み］のバグ報告票が本当に解決されたかは、テスターが確認するまで分かりませんから、［解決済み］のバグの数をZBBのに含めることもあります。このほか、開発者1人当たりのバグ数の平均や、1人当たりのバグ数の最大値、オープンされてから1週間以上未解決となっているバグの個数などもZBBの条件に含めることがあります。みなさんのプロジェクトに合ったZBBの条件を探してみましょう。

ZBBへの到達は、各マイルストーンのexit criteriaに含めます。つまり、各マイルストーンでZBBとなる日付はあらかじめ計画して決めておきます。この日に間に合うようにバグをクローズしていき、修正が間に合わなそうなバグはトリアージをして、そのマイルストーンに不要なものは次のマイルストーンやマイナーバージョンまで修正を延期します。そのマイルストーンのゴールに影響しないバグなら、起票した直後に次のマイルストーン以降にポストしてしまうのも良いでしょう。ただし、延期したバグの数が多すぎるときは、特別なアクションが必要になることもあります。

ZBBに到達した後は、トリアージ済みの(つまりトリアージチームに許可を得た)修正以外はコミットが禁止されます。この、修正のバーを上げてコミットを非常に難しくする期間をロックダウン(Lock Down)とか、その後さらにバーを上げてコミットをほぼできなくする期間をエスクロー(Escrow)などと呼ぶ組織もあります。このような期間を設けた後、修正が必要なバグがないことを確認し、そのマイルストーンをexitします。

最後のマイルストーンの末期では、ZBBの間隔が気になります。ZBBの間隔が短くなれば、リリース可能な時期が迫っていると考えられるからです。ZBBの状態を長く維持しようという目標が、チームをわかりやすく動機付けます。このようなイベントは、きちんとチームで共有しましょう。

|  |
| --- |
| To: 開発プロジェクトチーム  From: テストリード  Date: 2010/12/04  Subject: 最初のZBBに到達しました！  でへへ。本日、最初のZBBに到達したことを  ご報告します。  現在、次のような状況です。  オープンなバグ: 0件 > 48h  解決済みのバグ: 0件 > 48h  チームの誰が欠けても、達成しえなかった  素晴らしい成果です！  引き続き、よろしくお願いします。 |

# バーンダウンチャート

縦軸に残りのタスク数をとり、横軸に時間 (ビルド番号もしくは日付など) をとると、バーンダウンチャートというグラフになります。バーンダウン (burn down) とは、文字通り焼け落ちるという意味です。順調に開発がすすみ、残りのタスクの勢いも衰えてくれば、リリース可能な時期が近づいたと判断できます。

バーンダウンチャートは、スクラムというソフトウェア開発手法でバックログ (残りのタスク) を管理するために使われます。特に、スクラムでは開発期間を1ヶ月ごとの単位 (スプリント) に区切り、1ヶ月ごとにバックログをゼロにします。一定期間ごとに残りのタスクをゼロにする手法は、ZBBと同様です。スクラムでは、一ヶ月ごとにゼロにするバックログを指してスプリントバックログ、開発期間全体を通した残りすべてのバックログを指してプロダクトバックログといいます。必要に応じて、スプリントの期間内での実施が難しくなったタスクを、スプリントバックログの中から出してプロダクトバックログの中にポストします。これもZBBと同様です。ただし、スプリントバックログの中身は、そのスプリントの入り口で決め、スプリントの途中では追加しません。

このほかにも、有用な統計情報をさまざまな形でグラフにして表示できるBTSがあります。あるいは、バグ報告票データベースを.csv形式[[11]](#footnote-11)でエキスポートしたファイルをMicrosoft Excelで加工してグラフを書くのも良いでしょう。バグの数は、ビルドの健康状態を表す重要な指標です。ぜひ活用して、ビルドの健康管理に役立ててください。

|  |
| --- |
| **コラム バーンダウンチャートの中にバグ数は含めるか？** |

# コラム 開発中の催し⑤ バグバッシュ

|  |  |
| --- | --- |
| **コラム 開発中の催し⑤ バグバッシュ**  限られたリソース (テストチーム) だけで漏れなくテストを実施するというのは、非常に困難です。包括的なテストをするには、品質の良いテストを設計するという以外にも、猫の手を借りるとか、立ってる者は親でも使うといった工夫をするのも良いでしょう。  バグバッシュ (Bug Bash) とは、そんな工夫のひとつです。半日から2日間程度の時間を作って、プロジェクトメンバー全員でバグ出しをするイベントをバグバッシュといいます。Bash には、ジャパン・バッシングという言葉で聞くように、叩くという意味があります。テスト対象のソフトウェアを叩きまくって、バグを出すというわけです。が、Bash にはもうひとつ、楽しくどんちゃん騒ぎをするという意味もあります。バグバッシュでは、この意味が強調されます。つまり、バグバッシュとは、広い会議室を予約し、軽食を用意して集まり、楽しくどんちゃんとバグ出ししようというアイディアなのです。といっても、スマートフォンのような製品ならともかく、いちいち会議室を予約してテスト環境を構築するのも大変です。テストチームだけで、各自のデスクで粛々とテストするようなバグバッシュを企画してもいいのです。  では、バグバッシュとリグレッションテストで何が違うのかというと、リグレッションテストはテストケースに基づいてテストするのに対して、バグバッシュでは一般にテストケースを準備しないことです。ただし、効率よくバグ出しをするために、おおまかにテスト対象の機能の分担を決めて実施するのは良い習慣です。バグバッシュでバグを発見したら、それを再現できる手順を確立してバグ報告票を記述する必要があるのは、通常のテストと同様です。  最近バグがあまり出てこないとか、マイルストーンのexitが近いなどのタイミングでプロジェクトリードもしくはテストリードがバグバッシュを計画し、そのドライバーをアサインします。ドライバーは、多くの人に積極的に参加してもらえる方法を考えて、イベントをドライブします。おやつを準備するなどして、楽しいイベントにしましょう。  バグバッシュを、リグレッションテストと合わせて企画するのも良い習慣です。例えば、この週の3日間はリグレッションテスト、2日間はバグバッシュ、などのようにします。  イベントの後は、その成果をみんなで共有しましょう。バグバッシュのドライバーが、機能ごとやテスターごとに発見したバグ数をとりまとめてメールでチームに通知すると良いでしょう。   |  | | --- | | To: プロジェクトチームメンバー  From: バグバッシュのドライバー  Date: 2010/12/16  Subject: Bug Bash  みなさま  以下の要領でバグバッシュをやります！  場所: 各自のデスクで  日時: 2010/12/20 10:00 – 17:00  テスト対象: build#29 (前日の夕刻にビルド予定です)  インストール方法: …  バグの報告方法:  以下を、バグ報告票のテンプレートとしてください。  …  ご協力、よろしくお願いします。 | |

# バグ追跡システム (BTS; Bug Tracking System)

ここまで、ツールに依存しないバグ追跡の基本的な原則を紹介しました。紙と鉛筆を使う場合でも、ここに示した原則の多くは適用可能でしょう。しかし、本章の冒頭に述べたように、紙と鉛筆で書かれたバグ報告票を使うことは現実的ではありません。オープンなバグ数の推移のような統計情報を得るためにも、バグ報告票を電子化する必要があることはすぐにわかります。

そこで、バグの追跡にMicrosoft Excelのようなスプレッドシートが使われることも多いようです。しかし、スプレッドシートには、紙と鉛筆と同じく、バグ報告票それ自体を正しく追跡できないという問題があります。例えば、チームメンバーの誰かが、自分に都合の悪い記述をこっそり消してしまうかもしれません。あるいは、ファイルサーバで共有したエクセルファイルを大勢で同時に編集したら、ファイルが壊れてしまうかもしれません。このエクセルファイルはSCMリポジトリに入れて管理したとしても(すべきですが)、誰がいつどこを修正したかを追跡することはほとんど不可能です。エクセルファイルは、アプリケーション固有のバイナリファイルなので、簡単に履歴間の差分を確認できないからです。

バグ報告票の変更を追跡できなければ、リアルなバグの状態を追跡することもできません。そこで、役に立つのがバグ追跡システム(BTS)です。Bug Tracking Systemの訳語をバグ管理システムとする文章も多く見かけますが、私としてはバグ追跡システムとしてほしいところです。



図5 - 20 クライアントサーバ型のBTS

ほとんどのBTSはクライアントサーバ型のシステムです。最近人気のあるオープンなBTSにはTracやJira, Redmine, Mantisなどがあります。そのほとんどがWebクライアントサーバ型になっており、クライアントソフトウェアをインストールすることなく、Webブラウザをクライアントとして利用できます。また、そのようなBTSでも、EclipseなどのIDEをクライアントとして使うためのプラグインが用意されたものもあります。使い慣れたIDE(統合開発環境)に、その名の通り統合されたBTSクライアントは、とても便利なものです。

バグの追跡プロセスは、ソフトウェアの開発プロセスにおいて非常に重要です。実際に、バグの追跡プロセスをカスタマイズすることで、開発プロセスを改善できるほどです。このため、社内で使うBTSを自社開発するソフトウェア企業も多いようです。BTSのソースコードを持っていれば、そのBTSを自社の開発プロセスに合わせてカスタマイズできるからです。そのような BTS には、その企業のソフトウェア開発に関する知見やノウハウが詰まっているといえます。

これまでに私が所属した外資系のソフトウェア企業は、すべて自社開発のBTSを活用していました。しかし、現在は有償・無償を問わず、さまざまなBTSが利用可能になっており、その多くが十分なカスタマイズ機能を備えています。これからBTSを導入しようという人は、自分でBTSを開発する必要はないでしょう。

優れたBTSを使うと、ソフトウェア開発に多くの知見やインスピレーションが得られます。これらのBTSを使えば、みなさんの開発プロセスをドラスティックに改善できる可能性があるのです。ぜひ、みなさんの組織に合うものを探してください。



図5 - 21 BTSに登録したバグ報告票は  
さまざまな側面で分類して表示できる

# コラム オープンなBTSの名前について

|  |
| --- |
| コラム オープンなBTSの名前について  オープンなBTSの名前とその由来をいくつか紹介しましょう。  **Bugzilla**  オープンなBTSでは、最も古いもののひとつです。ブラウザFireFoxの前身であるNetscape Navigatorのバグを追跡するために開発されました[[12]](#footnote-12)。Netscape NavigatorのコードネームはMozillaといいました。Mozillaのバグを追跡するので、Bugzillaというわけです。これらは、日本の特撮映画「ゴジラ」にインスパイアされたものです。  **Jira**  AtlassianがリリースしているBTSです。Jiraは、もともと自社開発のBTSのコードネームでした。以前はBugzillaをゴジラと呼んで使っていた流れで、自社で開発したBTSをジラと呼ぶようになり、それがそのまま製品名になったのだそうです。  **Mantis**  Mantisとは、英語でカマキリという意味です。バグをたくさん捕えることを意図した、わかりやすいネーミングです。  **Scarab**  オープンなソフトウェア(Jakarta)で作られたオープンなBTSです。名前の由来となったスカラベ（フンコロガシ）は、不死のシンボルです。この昆虫は、掘った穴の中でフンに卵を産み付けて生涯を終えますが、次の年には生き返って（卵がかえって）穴から出てきます。死んだはずのバグがまた出てくるなんて、しゃれていませんね。 |

# BTSをタスクリストとして使う

BTSに登録したバグ報告票はプリオリタイズされ、各担当者にアサインされます。このため、バグ修正以外の作業もBTSにファイルしておけば、各メンバーの作業の見積もりや進捗をはかるのに大変便利です。大事な作業を忘れてしまうこともありませんし、忙しい人からそうでもない人に作業を渡すことも簡単です。また、バグの修正以外のタスクであっても、そのタスクの完了を確認して閉じるというワークフローをビルドのリリースというサイクルに合わせて行えるのは、ソフトウェア開発に大変都合が良いものです。BTSにファイルしたタスクは、どのような経緯を経て解決 (作業) されたかを後で調べる (たどる=追跡する) ことも可能になります。このように、解決すべき課題をすべてBTSにファイルしておけば、ソフトウェア開発の見通しがとても良くなります。



図5 - 22 BTSをタスクリストとして使う  
もちろん、自分でバグ(タスク)をファイルして  
自分をアサインしてもOK!

# BTSをハードウェア管理票として使う

BTSは、ハードウェアリソース (PCやモニタ) を管理するなどの目的にも使えます。PC1台につき1枚のバグ報告票 (ハードウェア管理票) を起票して、これをそのPCの所有者にアサインしておきます。ハードウェア管理票の件名には、当該のPCのモデル名やシリアル番号、資産番号などを記入しておくと良いでしょう。ハードウェア資産番号は、PCに物理的に貼り付けておきます。この番号のハードウェアを誰が使っているのか、BTSで追跡するわけです。BTSは、管理票の変更履歴を自動的に記録しますし、複数人で同時に閲覧したり編集するのが容易になりますから、スプレッドシートでPCを管理するよりもずっと扱いやすくなります。BTSは、ソフトウェア開発の現場には、非常に適用範囲が広いのです。

# バグ報告票データベースをサポートに活用する

もし製品のユーザーからバグの報告を受けたら、この製品を開発したときに使っていたバグ報告票データベースを検索してみましょう。同じバグがすでに登録されているかもしれません。

例えば、そのバグ報告票が「修正済み」となっていれば、このバグが修正されるビルド番号(バージョン情報)も書いてありますから、これをユーザーに知らせることができます。もし、修正されたビルド番号よりも新しいビルドをユーザーが使っていれば、一度修正したバグが再発したことがわかります。このリグレッションを修正するための重要な情報が、そのバグ報告票に記述されているかもしれません。そのバグ報告票をリオープンして［割り当て済み］の状態にまで差し戻せば、そのバグのこれまでの履歴も活用できますし、新しくバグ報告票を登録する手間も省けます。

あるいは、そのバグ報告票が「修正しない」という形で解決され、閉じられていれば、そのバグを「修正しない」ことにした理由や回避方法をユーザーに伝えることができるでしょう。

もし、同じバグがデータベースに見つけられなければ、そのまま新規にバグ報告票を起票します。このID番号をユーザーに伝えれば、ユーザーを安心させることもできます。このID番号が振られていることは、バグがきちんとハンドルされ、追跡されていることを示すからです。このように、バグ報告票データベースはユーザーサポートの際にナレッジベースとして活用できます。

ただし、開発プロジェクト用のBTSに、ユーザー対応の履歴情報を登録しすぎると、開発に必要な情報が取り出しにくくなりますし、プロジェクトチームとユーザーサポートの部署が分かれているときは、BTSに記述されている情報はsensitiveすぎて、サポート部に直接閲覧してもらうのが難しいこともあります。そこで、ユーザー数の規模が大きいときは、個別の案件(障害やクレームなど)の履歴を記録し、これを処理してクローズするための専用の追跡システムが必要となります。

このような、ユーザー対応の履歴を追跡するためのシステムを課題追跡システム (Issue Tracking System) といいます。チケットシステムとかインシデントシステムということもあります。このようなシステムに登録したチケットは、BTSのバグ報告票とは別の状態遷移をもちます。このチケットを有償として、ユーザーにはチケット (インシデント) を購入してもらい、ソフトウェア会社はこのチケットに対してサービスを提供することもあります。

また、ソフトウェア開発に特化した、プロジェクト管理などの機能を含む高機能なBTSを指してもITSということがあります。ITSには、バグ以外にも、新機能の提案やタスクリストなどが課題 (Issue) として登録され、追跡されるからです。また、その言葉のわかりやすさから、ソフトウェア開発のためのBTSやITSでも、そこに登録したバグ報告票 (Bug Report) や課題報告票 (Issue Report) を「チケット」とよぶものが増えてきました。

もし、ユーザー対応のためのチケットシステムを導入するなら、(運用上可能なら) ユーザー対応の履歴を記録したチケットと、BTSに登録したバグ報告票はリンクさせておきましょう。

BTSを二重化するポイントは、外部に公開するITSに、対応するプロジェクト内部のBTSの番号を記入 (その逆も) し、ふたつを対応づけておくことです。こうしておけば、実際にバグを修正・確認したとき、すぐにこれに対応するITSに「修正しました」と報告できます。

ウェブのシステムで実現されたITSなら、ユーザーにITSに直接起票したり閲覧してもらうことも容易でしょう。また、図5 - 21に示したITSはスプレッドシートで実現し、これをユーザーと共有することもあります。みなさんのプロジェクトに適切な方法を探してください。

# トヨタのかんばん方式と課題追跡システム

TPS (Toyota Production System; トヨタ生産方式) とは、ムダをなくすことで自動車部品の生産効率を向上させるための手法で、トヨタの自動車工場の現場で産まれました。TPSで重要な役割を果たすのが、かんばんという仕組みです。TPSでは、いくつかの種類のかんばんが使われます。例えば、共通の作業場所の壁に貼り出す作業指示かんばんがあります。これは、リーンソフトウェア開発といわれる手法で「ソフトウェアかんばん」として応用がなされており、すでにソフトウェア開発の現場に有用なものとして認知されています。リーンソフトウェア開発には、かんばん以外にもTPSを参考にしている部分があります。興味のある方は、ぜひ調べてみてください。

本書では、工場内を流通する仕掛けかんばんと、課題追跡システムにファイルした課題報告票との類似性を指摘したいと思います。仕掛けかんばんとは、その流通によって工場内の部品在庫を引き当て、部品在庫を極小化するためのものです。ある部品を必要とする消費部門では、生産依頼をかんばんという形で生産部門に渡します。生産部門はこれを受けて、かんばんと同じ数の部品を生産し、それにかんばんを貼り付けて消費部門に渡します。消費部門でその部品を消費したら、かんばんを剥がして繰り返し生産部門に渡します。すると、また必要なだけの部品を生産してもらえるという仕組みです。部品にかんばんを貼付してこれらを1対1で対応づけ、その数を追跡するわけです。かんばんには、対応する部品の品番と、その部品を要求した消費部門の名前が書かれています。また、同じ工場内で流通するかんばんには、色がついた無地の小さなプラスチック片が使われることもあります。



図5 - 24かんばんの流通と部品の引き当て

このようなかんばんの運用は、その枚数が肝心です。枚数が足りないと部品の引き当てが間に合いませんし、多すぎれば余剰在庫を工場内にかかえてしまいます。そこで、季節や時期に応じて枚数を適切に調整します。これを正しく行うには、「かんばんがなければ部品の生産を開始しない」というルールを守ることがポイントです。それが、余剰在庫となる部品を減らし、ジャストインタイムでの部品生産を可能にします。このかんばんは部品に貼付できるように、物理的に存在する必要があります[[13]](#footnote-13)。必要な分しか部品を生産しないので、もし不良品を消費部門に送ってしまうと、その工場のラインを止めてしまいます。そのような生産部門へのプレッシャーが、部品の品質を高めることにもつながったといいます。

では、課題追跡システムにファイルした課題報告票を考えてみましょう。課題報告票は、チーム内に流通して人的リソースを引き当てます。開発中のソフトウェアに内在するバグや課題は物理的には存在しないので、課題報告票はTPSのかんばんのように物理的に存在すべき理由はありません。電子的なものであっても、その状態と総枚数を追跡できればよいのです。これを優先度や各メンバーの負荷に応じて適切に割り振ります。これを正しく行うには、「課題報告票がなければ作業を開始しない」というルールを守ることがポイントです。



図5 - 25 課題報告票の流通と人的リソースの引き当て

図5 - 24では、人的リソースがチーム内を流通しているようにも見えますが、課題報告票がチーム内を流通していると解釈した方がわかりやすいでしょう。各課題には課題報告票が貼付されてその数と状態が追跡され、しかるべき担当者に送られるというわけです。もし、開発者が品質の良い解決を与えなければ、テストチームのテスト作業をブロックしてしまいます。

TPSの仕掛けかんばんは工場内での流通量 (全体の総枚数) を直接コントロールできるのに対し、課題報告票はチーム内での流通量をコントロールできません[[14]](#footnote-14)。バグが見つかれば、課題報告票 (バグ報告票) は増えてしまうからです。しかし、逆にいえば、チーム内に流通する課題報告票の状態と枚数で、現在のプロジェクトの状態を測れるということです。解決すべき課題をすべて発見してクローズしたとき、あなたは最終ビルドを手にしているはずです。そのためには、課題報告票を滞りなくチーム内に流通させることがとても重要です。課題をどこかで滞っていれば、メンバー間で課題報告票を受け渡して負荷分散したり、より適切な人に担当してもらうなどして、ボトルネックを解消します。





図5 - 26 かんばんの運用ルール





図5 - 27 課題報告票の運用ルール

このように、TPSのかんばん方式と課題追跡システムの運用には多くの類似点があります。大きな違いは、かんばん方式は後工程引取りといって、消費部門が必要な分だけを生産部門からプル (pull) してくるのに対し、ソフトウェア開発では欲しくもない課題が後から湧いて出てきてしまう (push) ことです。しかも、かんばんにはその解決方法 (どの部品を生産してどの消費部門に送るべきか) が最初から書いてあるのに対して、課題報告票が最初に起票されたときには、どのような経路をたどってどのように解決されるべきかはまだ書かれていないのです。それが、ソフトウェア開発の現場が混乱に陥りやすい原因のひとつかもしれません。

私は、これまでに所属した組織やプロジェクトで、それぞれ別のBTSやITSを使ってきました。その少ない経験に限っていえば、それらの運用はさほど違うものではありませんでした。しかし、課題報告票をアサインするポリシーが違うことはあったように思います。心地よく作業できるのは、課題報告票を開発者が自分で引き取って (pull) 自分をアサイン(サインアップ)するものでした。このような自律的な活動をメンバーに促すほうが、ソフトウェア開発は前に進みやすいものです。でも、それでは誰も手をつけないバグ報告票が発生しやすくなってしまいます。しかし、デブリードが開発者をアサイン (push) するようにすると、適切な担当者をアサインするのが難しかったり、開発者のやらされ感が強くなったりします。このため、この両方をうまく併用し、デブリードが課題報告票の状態をウォッチし、しばらく手つかずの報告票はうまく誰かにお願い (アサイン) していくのもひとつの方法です。

# まとめ

本章では、BTSを使ってソフトウェア開発を方向づける方法を示しました。このようなBTSを活用したソフトウェア開発は、決して新しいものではありません。定期的なビルドのリリースがソフトウェア開発を前に進める心臓の脈動だとすれば、バグ報告票はその血液のようなものです。血液が体に酸素を供給するように、バグ報告票はプロジェクトにフィードバックを供給します。心臓が止まれば、プロジェクトは死にます。どこかで血液の流れが滞れば、やはりプロジェクトは死にます。

私は、ITSにファイルした課題報告票 (Issue Report) を使ってソフトウェア開発を駆動するアプローチを、「課題駆動開発」(IDD; Issue Driven Development) と名付けてウェブの媒体に紹介したことがありました。私の文章が拙かったのか、残念ながらこの言葉は世間で認知されるには至らなかったようです。しかし、現在はITSをより上手に使おうという試みが多くなされており、特にITSに登録したチケットを使ってソフトウェア開発を駆動する「チケット駆動開発」(TiDD; Ticket Driven Development)というよりわかりやすい言葉が認知されつつあります。興味のある人は、ぜひ調べてみてください。

# 参考文献

## Bug management

<http://en.wikipedia.org/wiki/Bug_triage>

## Bug tracking system

<http://en.wikipedia.org/wiki/Bug_tracking_system>

## Issue management

<http://en.wikipedia.org/wiki/Issue_management>

## Issue tracking system

<http://en.wikipedia.org/wiki/Issue_tracking_system>

## Comparison of issue-tracking systems

<http://en.wikipedia.org/wiki/Comparison_of_issue_tracking_systems>

## Software Regression

<http://en.wikipedia.org/wiki/Software_regression>

## Regression Testing

<http://en.wikipedia.org/wiki/Regression_testing>

## Bug bash

<http://en.wikipedia.org/wiki/Bug_bash>

## Bugzilla

<http://ja.wikipedia.org/wiki/Bugzilla>

## Trac

<http://ja.wikipedia.org/wiki/Trac>

## Redmine

<http://ja.wikipedia.org/wiki/Redmine>

## Jira

<http://ja.wikipedia.org/wiki/JIRA_(ソフトウェア)>

## Mantis

<http://www.alles.or.jp/~sogabe/mantis/>

## アート・オブ・プロジェクトマネジメント

<http://www.amazon.co.jp/dp/4873112990>

## Ship It! ソフトウェアプロジェクト 成功のための達人式ガイドブック

<http://www.amazon.co.jp/dp/4274066568>

## トヨタ生産方式―脱規模の経営をめざして

大野 耐一 (著)

<http://www.amazon.co.jp/dp/4478460019>

## トコトンやさしいトヨタ生産方式の本

トヨタ生産方式を考える会 (著)

<http://www.amazon.co.jp/dp/4526052450>

## みんなのシネマレビュー

<http://www.jtnews.jp/>

## みんなの意見は案外正しい

James Surowiecki (著), 小高 尚子 (監修, 翻訳)

<http://www.amazon.co.jp/dp/4042977014>

1. 筆者は、[終了] という語には閉じるという意味が含まれないので、[開始]の対義語としては不適切と思いますが、これに代わる良い訳語を見つけられませんでした。Open/Closedを、開店/閉店のように訳せれば良いのですが。閉了や閉終、閉票などという語は手持ちの辞書に見つかりませんでした。もっとも、本書は理解を助けるために邦訳の語を載せましたが、私はBTSの運用でこのような日本語を使ったことはありません。みなさんの多くもそうだろうと思います。 [↑](#footnote-ref-1)
2. バグ報告票に限らず、一枚の紙にはひとつの内容だけを記入すべきルールを一件一葉といいます。蛇足ですが、ひとつの文にはひとつの内容だけを記入すべきルールは一文一葉といいます。 [↑](#footnote-ref-2)
3. もちろん、最後のは冗談です。でも、あまり笑えません。 [↑](#footnote-ref-3)
4. 私の手元にある複数の英和辞書には、severityの訳語として「重症度」「過酷度」「発病度」などが見つかりますが、「重要度」はありませんでした。 [↑](#footnote-ref-4)
5. コミットコメントについては、★表3-4を参照してください。 [↑](#footnote-ref-5)
6. 海外のBTSやテストケース管理ツールの日本語版には、リグレッションバグの訳語を「回帰」とするものがありますが、適切な訳語ではないと思います。「リグレッション」で良いのではないでしょうか。せめて「退化バグ」とか「後退バグ」としてほしいものです。 [↑](#footnote-ref-6)
7. このことを教えてくださった岡村敦彦さんに感謝します。 [↑](#footnote-ref-7)
8. Trivialとは、些細な、つまらない、といった意味で、深刻度を表す基準などに使われることがある語です。トリビアルとトレビの泉をかけた、トリビアの泉というテレビ番組が一世を風靡したこともありましたね。 [↑](#footnote-ref-8)
9. BTSに登録された新機能の提案は、feature workとかenhancement requestなどといいます。これを正式なドキュメントに起こしたものをRFP (Request for Proposal) とかDCR(Design Change Request)などといいます。 [↑](#footnote-ref-9)
10. ここでいう「収束」とは、数学的に定義されたものではありません。 [↑](#footnote-ref-10)
11. Comma Separated Value; カンマ区切りのテキストデータファイルのことです。 [↑](#footnote-ref-11)
12. ★★余談ですが、Netscape Navigatorの前身は、世界最初のブラウザであるMosicでした。 [↑](#footnote-ref-12)
13. 遠くはなれた工場間でスムーズにやりとりができるように、電子化されたかんばんもあります。電子化されたかんばんは消費部門から生産部門にネットで送られ、生産部門で印刷されて部品に貼付されます。 [↑](#footnote-ref-13)
14. 蛇足ですが、プロジェクトの途中で、チーム内の人的リソース(メンバーの人数)を調整・増減させるのも難しいものです。 [↑](#footnote-ref-14)