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、一般に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管理することになります。バグ報告票は紙に手書きで書いても管理しきれません。開発規模が小さければ、Microsoft Excelのような表計算ソフトウェアを使ってもいいかもしれません。しかし、複数人でより効率よくバグを追跡するには、バグ追跡システム (BTS; Bug Tracking System) あるいはより一般に課題追跡システム (ITS; Issue Tracking System) というソフトウェアを使うべきです。BTS にされたバグや課題は、チームメンバ各人に対してタスクリストのように扱うことができ、非常に便利です。これはいろんな点でメリットが多いため、BTS にはバグだけでなく、これから実装すべき新機能のほか、さまざまなタスクなども一緒に課題として登録することが一般的です。本章では、発見したバグや課題を追跡し、これを解決することによって開発プロセスを駆動するアプローチについて説明します。

# バグ追跡システム

バグの追跡プロセスは、開発プロセスにおいて非常に重要なものです。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。正直にいえば、私は業務ではオープンなBTSを使った経験がありません。これまでに私が所属した外資系のソフトウェア企業は、すべてインハウスのBTSを活用していました。しかし、もしみなさんが独自のBTSを持っていなくても、TracやJira, Redmine, Mantisなど、無償で利用できるオープンな BTS がたくさんあります。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みてください。

BTSとは、今風の言葉でいえば「プロジェクト見える化ツール」ともいうべきものです。バグ以外にもソフトウェア開発に関するさまざまな課題をファイルすることができ、それらをさまざまな側面で分類して表示します。複数人で同時に使うことも簡単で、各人に割り当てた作業を優先度順に表示したりできます。

BTSに登録されたバグや課題は、それをアサイン[[1]](#footnote-1)された担当者が１人だけで解決することはできません。バグ報告票ごとに段階を踏みながら、複数のメンバー間で課題をやり取りし、バグが解決されたと確認した段階でこれを閉じます。BTSは、このワークフローのプロセスを定義し、管理します。BTSを介して、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ることができます。



図5 - 1 バグ報告票がチーム内を流通しながら  
解決されていくようす



図5 - 2 BTSに登録した課題は、さまざまな側面で分類して表示できる

表5 - 1から表5 - 4に、バグを一覧表示するBTSの各ビューの例を示します (煩雑になるのを避けるため、各バグ報告票の件名は一部を省略しました)。このほか、自分に割り当てられたバグだけを優先度順や状態別に表示したりできます。

表5 - 1 BTSの状態別ビューの例



表5 - 2 BTSの優先度別ビューの例



表5 - 3 BTSの担当者別ビューの例



表5 - 4 BTSの解決ビルド#別ビューの例



表5 - 4に補足します。このビューは、開発作業の実績を表すとともに、ビルド計画も同時に表しています。例えば、先週にテストチームにリリースした最新のビルドの番号が#38であり、ビルド#39とビルド#40はそれぞれ来週と再来週にテストチームにリリースする計画であるとします。すると、表5 - 4の上半分 (build#38以前) は実績であり、下半分 (build#39以降) は計画ということになります。各バグについて、それぞれを担当する開発者がどのビルドで解決 (修正) できそうかの見通しを書き込んでもらえば、このような管理ができます。

そこで、今週リリースした最新のビルドの番号は何番か、来週にリリースするビルドの番号は何番か、それぞれのビルドの状態はどうなっているか、このようなビルド計画をチーム内で共通認識としておくことが重要となります。このため、BTSによっては、ビルドノート (リリースノート) を一緒にBTSに登録して、各ビルドの状態 (開発中やリリース済みなど) を追跡し、これをバグ報告票と関連づけておけるものもあります。ビルドの状態については、★図2-9を参照してください。

# バグとは

バグの追跡方法を説明する前に、バグについて知っておきましょう。バグとは、ソフトウェアが意図もしくは期待したとおりに動作しない不具合のことです。ソフトウェアが身の周りにあふれている現代、バグは命にかかわる問題になり得ます。2001年、ディスカバリー号に搭乗していた4人の宇宙飛行士は、この船に搭載されていたコンピュータHAL9000に潜んでいたバグが原因で命を落としました。というのは1968年に書かれた小説の話ですが、最近もトヨタ自動車のリコール問題などが現実に起きています。ソフトウェアのバグが人命さえおびやかす可能性はとても大きくなっているのです。**(本節、要検討)**

とても身近なところでは、バグは私たちソフトウェア開発に携わる人たちの貴重な時間を搾取しています。ソフトウェア開発はバグとの戦いであり、バグを退治してこの戦いに勝利することが、ソフトウェア開発プロジェクトにとって重大な関心事です。

# 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[[2]](#footnote-2)などがあります。 |

# バグの発見と文書化

ソフトウェアの中に潜在的に含まれているバグは、これをすべて発見し、文書化してBTSにファイルする必要があります。この文書のことをバグ報告票 (Bug Report)、あるいはより一般に課題報告票 (Issue Report) といいます。最近のオープンなBTSでは、TracやRedmineなどのように、これをチケット (Ticket) と呼ぶものも増えてきました。

一般に、文書化して管理しなければならないのは、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) 統合ビルドのバグだけです。開発者がサンドボックスでビルドしたビルドで見つけたバグは、文書化する (BTSにファイルする) 必要はありません。というのは、そのバグはその開発者のサンドボックスでだけ発生するものかもしれないからです。開発者は、そのバグをいちいち文書化することなく、次のビルドまでにこっそり治してください。ただし、このバグを含むソース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同じバグがあると考えられる場合や、すぐに修正できるめどが立たない場合など、そのバグの再現手順にテストケースとしての価値が認められることもあります。このようなときは、そのバグがリリース済みの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BTSにファイルしておくとよいでしょう。

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、テストチームによるテストが実施されないように、開発者がリリースノートに制限事項として文書化しておきます。これにより、既知のバグを、それを知らないテスターが再発見して文書化してしまう無駄を避けられます。開発者には、未実装の機能や既知のバグをテストチームに伝える義務があります。この義務を果たすために、開発者とテスター間のコミュニケーションツールとして、リリースノートを利用します。ただし、テストチームにより未実装の機能がバグとしてBTSにファイルされてしまっても、開発者はそれほど困りません。検証可能な仕様 (テストケース) を段階的に追加・詳細化してもらったのですから、むしろ感謝すべきことです。ある機能が未実装であることを確認する手順を先に記述しておくことは、テストファーストという観点からも有用です。もちろん、開発者が自分で未実装の機能をBTSに登録しておいても構いません。

バグを発見するための具体的なテスティングのプロセスは、本書では説明しません。モデリングの技術が開発者の職人技であるのと同じく、テスティングの技術はテスターの職人技に依存するものであって、複数のロールから構成され、属人的なものを排した開発プロセスとは直交するものだからです。もちろん、モデリングがそうであるように、テスティングもソフトウェア開発におけるアクティビティの1つですが、その具体的なの技は本書で説明すべきことではなさそうです。これらについて詳しく知りたい方には、テストの技法に関する書籍の一読をお勧めします。

# 一件一葉の原則

ひとつのバグ報告票には、ひとつのバグだけを記入します。これを一件一葉の原則といいます[[3]](#footnote-3)。例えば、ある事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します。そうでないと、開発者は記入されているバグ全てを修正してからでないと、そのバグ報告票を閉じることができず、タスクの管理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。

また、親子関係にある別のブランチに同じバグが含まれることもあります。このときは、同じバグについて、ブランチの数だけバグ報告票を起票することを勧めます。このようにしておけば、当該のブランチで修正をコミットしたタイミングでバグ報告票を解決済みにできるからです。つまり、修正が終わっていないブランチについては、対応するバグ報告票をオープンのままにしておけます。

逆に、すでに起票済みのバグについて、それに気づかずに別のバグ報告票を起票してしまうこともあります。これは好ましくないので避けるべきです。具体的には、バグ報告票を起票する前には、BTSデータベースを検索して、報告しようとしているバグがすでに起票されていないかどうか探してみるべきです。重複したバグ報告票を起票してしまった場合の処理については後述します。

# バグ報告票のライフサイクル

図5 - 2に、バグ報告票のライフサイクルを示します。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。この状態は、バグ報告票の項目として各報告票に記載され、BTSで管理されます。この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のが図5 - 3です。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、これをプロジェクトマネージャやデブリード、あるいはトリアージチームの責務とすることもあります。



図5 - 3 バグ報告票の状態遷移

表5 - 5 バグの状態と、その担当者

|  |  |  |
| --- | --- | --- |
| 状態 | 状態の別名 | 担当者 |
| 新規 | New, Triage Neededなど | デブリード、  テストリード、  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) |



図5 - 4 バグ報告票を閉じる手順

バグ報告票を新しく作成して BTS にファイル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、バグ報告票を作成できる人を テスターだけに制限することもあります。多くの BTS は、このような柔軟なアクセス制御機能をもっています。

それぞれのバグ報告票は状態をもち、この状態に応じてその担当者が決まります。最初に記述されたときの状態は［新規］です。以下に、バグ報告票の状態と、各状態における担当者を列挙します。

|  |
| --- |
| 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、多くの場合で、バグ報告票が［解決済み］に遷移したときの担当者 (テスター) をこの段階で同時に割り当てておきます。特にテスターをアサインしない場合、このバグを報告した人がこのバグのテストも担当することに決めておくとよいでしょう。というのは、このバグを報告した人が、最も適切にこのバグが解決されたかどうか判断することが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のように、バグを発見した人とバグ報告票を記述する人が異なるときもあります。そのようなときは、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するようにします。

## ［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「修正済み」という形で解決するときには、3章で示した手順でソースファイルをリポジトリにコミットし、そ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て、この状態を［解決済み］とするわけです。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。

バグ報告票を［割り当て済み］にするとき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。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」という形でバグ報告票を［解決済み］にしたときは、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はいないでしょう。これを担当するテスターは、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、当該のビルドがリリースされたらすぐにテストが開始できるように準備しておきます。このビルドがリリースされたら正しく修正されているかどうかをテストして確認し、確認に使ったビルドの番号やテストの手順などをこのバグ報告票にコメントして、その状態を［確認済み］に変更します。開発者が選択した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すことができます。

## ［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の状態を［終了］にしなければいけません。

ただし、［終了］となったバグ報告票でも、テストケースとして価値のあるものですから破棄してはいけません。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、ユーザーからバグの報告を受けたときにBTSを検索すれば、そのバグがすでに起票済みで、しかもすでに［終了］になっていることを見ることもあります。その場合、もしユーザーが古いバージョンのソフトウェアを使っていたら、この問題が修正済みのバージョン番号をユーザーに知らせることもできます[[4]](#footnote-4)。このように、BTS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。

BTSを検索したところ、同件と思われるバグがすでに報告されており、すでに「修正済み」という形で解決されていたとします。ところが、ユーザーがもっと新しいビルドを使っていたなら、以前修正したはずのバグが再発したことになります。このバグ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。このようなときは、バグ報告票を再開して［割り当て済み］の状態にまで差し戻します。BTSは、各バグの履歴を記録して追跡するためのものですから、［終了］となったバグ報告票であっても破棄してはいけません。

# 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」には、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はない」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) 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 - 6 バグの解決方法 (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・検証しなければなりません。修正が確認できれば、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)、その理由を追記して開発者に差し戻します(バグ報告票の状態は［割り当て済み］に戻ります)。同意できれば、テスターはこのバグ報告票の状態を［確認済み］とします。そもそも、すでに報告済みのバグについて、重複したバグ報告票を起票してしまうのは、みっともないので避けるように気をつけるべきです。

## 「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せます。同意できれば［確認済み］にします。

## 「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」 (Defer)

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)

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。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。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。多くのBTSは、バグ報告票の書式をカスタマイズし、項目を増やしたり減らしたりする機能をもっています。

表5 - 7 バグ報告票に記載すべき項目と別名 (例)



## バグ報告票のID番号(ID)

ユニークなIDを記入する項目です。ほとんどのBTSは、この項目を自動的に埋めてくれます。バグを一意に特定できるように、バグ報告票には通しの番号を付けます。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の文書には、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をすることもできます。ただし、IDの書式をカスタマイズできるBTSは少ないようです。

このIDにより、バグ報告票はほかのバグ報告票 (またはその他の文書)から参照できるようになります。例えば、バグ報告票が重複して書かれてしまったとき、開発者は最初のバグ報告票のID番号だけをそのバグ報告票に追記し、「重複」という形で解決済みとします。また、ユーザーから報告されたバグの場合には、起票したバグ報告票のID番号をユーザーに伝えることで、ユーザーに安心してもらえます。このID番号が振られていることは、きちんとバグがハンドルされ、追跡されていることを示すからです。

## 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。もっと具体的な内容を記述してください。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、利用するBTSにバグ報告票をグループ化する機能がなければ、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という決まりを作って、関連するバグ報告票を簡単に検索できるようにするための工夫をすることがあります。

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

どんなことを書くべき項目かは、説明の必要はないでしょう。ただし、手間をかけずに報告票に記入できる工夫が、より必要となる項目です。数百～数千ものバグ報告票に、同じこと(製品名)を手書き(手入力)するのは、大いなる工数の無駄遣いです。気の利いたBTSは、製品名などのよく使う項目値を含むテンプレートを適用することで、それらを自動入力する機能を備えています。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ておきます。

## 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。BTSによっては、階層構造をしたサブシステム名を管理できるものもあります。

## バグの種別 (Issue Type)

このバグ報告票 (課題報告票) 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 - 8 バグの種別の選択肢 (例)

|  |  |
| --- | --- |
| Issue Type | 意味 |
| Code Defect | バグ、不具合 |
| 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 (Developer/Tester)

このバグを担当する担当者たちの名前を記入する項目です。バグ報告票は、その状態により、これを担当するチーム(ロール)が決まります。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。文書の状態が変わるたびに、”担当者”という項目を書き直すのは良くありません。バグ報告票には、あらかじめ ”開発担当” “テスト担当” といった各担当者を記述する項目を用意しておき、バグ報告票の状態遷移とともに現在の担当者も遷移する、といった理解をすべきです。

## 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5-5節と5-6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よいでしょう。品質の高いバグ報告票には、必ず ”いつも再現する” と書かれるべきです。”なかなか再現しない” などと書かれたバグ報告票は価値が低いものです。そのまま開発者に「再現しない」という形で突き返されてしまうかもしれません。また、”ときどき再現する” というのは、マルチスレッドを利用したソフトウェアなどでは実際にあるため、書かれるべきでないとはいい切れません。しかし、再現のための条件を絞り込めていないわけですから、やはり ”いつも再現する” と記入されたバグ報告票よりも相対的に品質が低いものです。再現性が良いバグは、修正も容易です。

この項目が用意されていないBTSでは、再現手順の項目などに ”再現性: 3/5” などとして付記しておくと良いでしょう。

## 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、バグの再現性が悪いときにはぜひ添付したいところです。そのような証拠があれば、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。

## 深刻度と優先度 (Severity / Priority)

深刻度は、このバグがどれだけ深刻かを示すもので、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です[[5]](#footnote-5)。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)。Severity (シビリティ) とは、そのバグがどれだけシビアなのか、ということなのです。重要度といってしまうと、そのバグが再現する操作手順やシナリオがユーザーにとってどれだけ重要なのか、そのバグ報告票の再現手順 (テストケース) としての側面に注目してしまいがちですが、このような再現手順の重要性は再現するバグの深刻度とは切り離して、トリアージやプリオリタイズ、リグレッションレベルの設定時などに考慮すべきものです。

優先度は、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、優先度をつけて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 - 9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。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、BTS 上でオープンされているすべてのバグの深刻度が同じく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。

深刻度と優先度は、トリアージの際の重要なインプットとなります。トリアージについては後述します。

表5 - 9 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 | あったらうれしい |
| 8=Documentation | ドキュメント (ヘルプ) の誤り |

また、深刻度と再現性もきちんと区別してください。「これは再現頻度が高いから高い深刻度をつけておこう」などということはせず、深刻度と再現性は分けて考えましょう。

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、といった絶対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 - 10 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。ユーザーが商品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、点数の基準を表示す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、適切な絶対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に示します。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BTSから収集できるようになります。  表5 - 11 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 - 12のようなものが適当です。トリアージについては、後の節で改めてとりあげます。

表5 - 12 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 - 13 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 | ユニットテストで |
| 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。

## 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、それを再現できた (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、テスト結果を次のように記入しておくと良いでしょう。

|  |
| --- |
| **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｛する|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) する予定のビルド番号を記入する項目です。このバグが修正される前 (バグ報告票の状態が［割り当て済み］のとき) は、この項目は予定を表します。しかし、開発者がこのバグを実際に修正し、バグ報告票の状態が［解決済み］に遷移したとき、この項目は予定ではなく実績を表すことになります。詳しくは、表5 - 4を参照してください。

開発者は、アサインされたバグ報告票のこの項目を早めに記入することで、修正の見通しをほかのメンバーに伝えられます。必ずしもそうすべきというわけではありませんが、バグを修正してこのバグ報告票の状態を［解決済み］にするときには、この項目の記入は必須となります。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始めるからです。この項目が埋まっていなければ、テスターはテストを開始できません。逆に、早い段階でこの項目が埋まっていれば、テスターも早い段階でこのバグの修正確認のための計画や準備を始められます。

## 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。

ただし、この項目はバグ報告票の書式としては用意しないこともあります。というのは、すべてのバグが「修正済み」という形で解決されるとは限りませんし、このバグ報告票の再現手順をテストケースとして別のビルドに対して繰り返し実行することがありますから、この項目をひとつだけ用意しても、複数のビルド番号を書ききれないからです。このように、この項目がバグ報告票の書式に用意されていないときは、コメントの項目などに「build#23で修正を確認しました」などと記入しておきます。

## 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履歴欄に記録してくれるBTSもあります。

## 修正時のコミット番号 (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[[6]](#footnote-6)。また、そのコミット番号 (チェンジセット番号) をバグ報告票に記入します。これにより、このバグ報告票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報告票を解決したものか、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。

## 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。この再現手順は、テスターがテストケース管理ツールに転記して管理されることもあります。

## 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:**  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。どこがいけないのでしょうか。

答えは、最後の3.のステップにあります。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そうなっていない、という意味なのか、「会議室が予約されました」と表示してはいけないのに表示してしまっている、という意味なのか、分かりません。

これでは、このバグ報告票をアサインされた開発者が自分の環境でこの手順を試しても、どう修正すべきなのかわかりませんし、そもそもバグを再現できているかどうかさえ判断できません。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:**  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 を参照のこと。 |

## 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。この情報も、開発者がバグの原因を突き止めるための良い材料となります。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。ユーザーは、回避方法がある既知のバグ (known issue) は許してくれることが多いものです[[7]](#footnote-7)。もし回避方法を発見したなら、忘れず記入してください。



図5 - 5 バグ報告票を書くためのステップ  
(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

★ページの図2-5ビルドの世代交代を、図5 - 5に再掲します。このように、テストチームによるテストと開発チームによる開発は、同時並行して行われます。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 - 6 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。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。品質の良いバグ報告票が、段取りの良いソフトウェア開発のために、非常に重要であることがお分かりいただけるでしょうか。また、開発チームとテストチームの責務についても、この図で整理できるはずです。



図5 - 7 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。私がよく見かける間違いを以下に紹介しましょう。

## 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、Microsoft Excel でバグを追跡するときや、よくできた BTS を使った経験がない人は、状態と処理方法を区別しない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。必要に応じて、開発者が選択できるバグの処理方法は開発途中で増やしても構いません。しかし、バグ報告票が取り得る状態を追加するのは、慎重に行ってください。図5 - 3に示したバグ報告票の状態遷移をよく眺めてみてください。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。もちろん、開発するソフトウェアのドメインによってはこの状態の種類をカスタマイズする必要もあるでしょうが、状態と処理方法は明確に区別されるべきです。

## バグ報告票の妥当性を検査するプロセスがない

バグ報告票を記述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ね。このようなモデルをV&V (Validation & Verification) ということがあります。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。この問題に対するもっとも簡単な解決は、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、としてもよいでしょう。

## バグ報告票の解決を確認するプロセスがない

前述した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。つまり、バグの状態が［解決済み］から［確認済み］に遷移しないということです。開発者により解決されたと報告されたバグは、本当に解決したのか、テスターが実行可能なビルドを使って確認しなくてはいけません。このプロセスが欠けていると、バグの数はいつまでたっても収束しません。バグの解決とその確認は、開発者とテスターの二人三脚で行われるべきです。

## ビルド番号が管理されていない

実行可能なソフトウェアにビルド番号を付加していない事例は非常によく見ます。ソフトウェアテストは、その結果をきちんと文書で残す必要があるとき、そのテスト対象も明確に記録しておかなくてはいけません。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。テスト対象を一意に特定できるように、ビルドには必ずビルド番号を付加してください。

## 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。例えば、次のようなやりとりを考えてみましょう。



図5 - 8 バグ報告票だけで、やりとりをすると…

開発者とテスターが顔を合わせず、バグ報告票だけでやり取りをすると、これだけのコミュニケーションのために、ビルドを何度もリリースしなければならなくなります。いったい、このバグを解決するまで何回ビルドをリリースしなければならないのでしょうか。これでは解決できるどころか、最終ビルドを手に入れる前に、人間関係が崩壊してしまいます。開発者は、テスターにより報告されたバグが「バグでない」と考えるなら、まずその報告者であるテスターに対して「これこれの理由で、このバグを修正せずに［解決済み］とするけど、いいかな？」と確認を取るべきですし、テスターが開発者の解決に同意できないときにも、いきなりバグ報告票をつき返すのではなく、同様の丁寧さ、慎重さが必要でしょう。

## 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ておくべきです。

## テスターのモチベーション管理に失敗する

テストはエンドレスな作業ですから、テストチームのモチベーションを維持するのは困難です。「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

例えば、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、これを昇給やボーナスの査定に使う組織があります。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、些細なバグが大量に重複して報告されてしまうこともあります。発見したバグ数が多かったテスターをみんなで称えるのは、仕事を楽しくし、やる気を向上する上で大きな効果があります。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。日本では、これをデグレードというソフトウェア会社も多いようです。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。しかし、リグレッションという方が、以前動いていたものが動かなくなる、というニュアンスがはっきりするように思います。Regressionとは退化とか後退、後戻りといった意味だからです。また、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。



図5 - 9 リグレッションバグ

動いているソフトウェアに気軽に手を入れると、リグレッションを発生させてしまう危険があります。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なければいけません。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すると、それを修正する時間は残っていないかもしれないからです。そもそも、発生したリグレッションを発見することさえ、でき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プラクティスです。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) 十分な数の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。ソフトウェア開発の末期段階では、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もう1度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。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。

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を使ってリグレッションテストを実施することを、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する必要があります。自動化していないテストをリグレッションするには大変な工数がかかるので、そ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することもできます。これにより、リグレッションが発生しても、これ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。

## フルリグレッションテストの実施は、ユーザーにリリースする直前だけにする

フルリグレッションテストを実施した後にコードを修正したら、フルリグレッションテストはもう一度最初からやり直しとなってしまいます。このような不幸を避けるには、フルリグレッションテストはコードフリーズが宣言された後、ユーザーリリースの直前にだけ実施するようにします。もし、このテストでバグを発見したら、それは修正しないままリリースすることも視野に入れて検討します。コードを修正しなければ、フルリグレッションテストを実施し直す必要もないからです。この判断については次節で説明します。

## 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、各テストケースの必要性や実施の容易さなどから、各テストケースにリグレッションレベルを設定しておき、今回のリグレッションではレベルいくつのものまで実施、といった管理をするとよいでしょう。短時間で実施できる重要なシナリオやケースを絞り込み、このリグレッションレベルにゼロを設定すれば、毎日実施するスモークテストのためのテストスイートも管理できます。また、リグレッションレベルは適宜見直すと良いでしょう。コードの修正がほとんど入らなくなった安定した機能をテストするケースはリグレッションレベルを落とし、逆に不安定な機能をテストするケースのリグレッションレベルは上げていくことができます。

ただし、この方法は必要なテストの実施まで省略してしまう可能性があるので注意が必要です。リグレッションの発生は、ある修正をしたときに、およそ予期しなかった部分へ影響が及んでしまうことによります。「この部分の修正は、この部分にしか影響がないから、この部分だけテストすれば大丈夫」という思い込みが、リグレッションを生みます。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。

# 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。

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。最終ビルド候補のリグレッションテストで発見したバグは、なるべくなら修正の対象としたくはありません。修正してしまえば、リグレッションが発生してしまうかもしれません。リグレッションが起きなかったことを確実にするために、次の最終ビルド候補でフルリグレッションテストをやり直す必要があります。その結果、やはり新しいバグを埋め込んでしまったことが判明したら、これを修正して最終ビルド候補をリビルドし、またフルリグレッションテストを全部やり直すというとんでもない工数が掛かります (さらにその修正により、その次の最終ビルド候補に、また別のバグを埋め込んでしまったら？)。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 - 10 医療で使われるトリアージタグの例 (Wikipediaより)

# トリアージとプリオリタイズの違い

開発の末期段階で山のようにバグが積みあがっていれば、それらを正しく優先順位づけて、修正すべきものから修正すべきであるという考え方はわかりやすいかもしれません。しかし、トリアージはそのような優先順位づけ (プリオリタイズ; prioritize) とは違います。残ったバグに優先度をつけて上から順に処理していき、単純に治し切れなかったものはそのままリリースする、というのはトリアージではありません。ソフトウェアにおけるトリアージでは、優先度をつけることよりも「治すべきか治さざるべきか」という判断がとても重要です。

例えば、開発の末期段階でひとつだけバグが残ったとしましょう。ひとつだけですから、優先順位をつける必要はありません。リリースはあと1週間後に迫っています。開発者の１人は、恐らく3日あればこのバグを治せるだろう、と言っています。果たして、このバグは治すべきでしょうか？いいえ、仮にそれを1日で修正できたとしても、その修正をリポジトリにコミットしていいかどうかは別の問題です。



図5 - 11 治すべきか、治さざるべきか？

図5 - 11に示したように、多くのことを勘案してトリアージをする必要があります。この図の左側は分かりやすいと思いますから、右側だけを次節で説明しましょう。

# トリアージ基準

がんばってバグを治しても、逆にソフトウェアがしてしまうことがあります。トリアージをするときには、次のことを考慮してください。

## テスト工数へのインパクト

実際にコードを修正して、それをリポジトリにコミットしたら、その修正が正しいかを確認するためのテストが必要になります。場合によっては、新しいテストケースを追加したり、既存のテストケースを修正したりする必要も生じます。これがテスト工数へのインパクトです。

さらに、開発の末期段階では、その修正が正しいかどうかをテストするだけでは足りません。修正の種類によっては、フルリグレッションテストをやり直す必要が生じます。ただし、実際に最終ビルド候補を何度もビルドし直して、そのたびにフルリグレッションテストをするのはあまり現実的ではありません。リリース直前では、テストが必要な範囲が明確なためにフルリグレッションテストをしなくても済む、つまりテスト工数へのインパクトが限定的な修正だけがコミットできます。



図5 - 12 テストへのインパクトの大きさ

## リグレッションリスクの大きさ

リグレッションが発生するリスクをリグレッションリスクといいます。一般に、リグレッションリスクの大きさは図5 - 12に示した要素で見積もることができます。



図5 - 13 リグレッションリスクの大きさ

コード変更の規模が大きかったり、複雑なコードに手を入れたりすると、リグレッションが発生しやすくなります。また、新機能追加のために、しばらく修正していなかった (安定した) コードに手を入れるときにも、リグレッションが発生しやすくなります。

## ユーザーへのインパクト

当該の修正をすることで、ユーザーに影響が及ぶ可能性があります。大きく分けて、修正による直接的な影響と、その修正によるリグレッションをユーザーに漏らしてしまったときの影響があるでしょう。

前者では、例えばテストをやり直すためにリリースの時期を遅らせなければいけなくなることがあります。また、リグレッションリスクを小さくするために大がかりな修正を避けた場合は、その修正によってソフトウェアの操作方法が変わるなど、何らかの制限事項が生じることもあります。

後者ですが、その修正を入れることで、実際にリグレッションが発生するかどうかはわかりません。万が一発生したとき、どの程度深刻なリグレッションがどの程度の頻度で再現することになるのかも、なかなか見積もれるものではありません。



図5 - 14 ユーザーへのインパクトの大きさ

しかし、エンドユーザーの人数や、ユーザー業務の種別はトリアージのときに考慮できます。例えば、その製品を使うユーザーが数万人いるとか、その製品が医療機器や金融システムの一部であったりする場合には、開発の末期段階でのバグの修正はより慎重にならざるを得ません。ユーザーへのインパクトが大きく、またテストの時間を十分に取れないなら、やはりリリース直前の修正はあきらめるべきです。

テストインパクトとリグレッションリスクについては、開発者がコードを修正できたとき、コードレビュー依頼票に記入すると良いでしょう。それらがあまりに大きいときは、コードをコミットする前に開発者もしくはレビューアがトリアージチームにアラートを投げます。コードレビュー依頼票の書式については、表3-4を参照してください。ユーザーインパクトは、必要に応じてトリアージ担当者が考慮します。

以上のように、開発の末期段階では、開発者がバグを修正した後、もう一度トリアージのプロセスを踏んで、その修正をリポジトリにコミットすべきか否かを判断することがあります。トリアージのインプットとなる当該の修正によるリスクやインパクトは、修正前よりも修正後の方が正しく見積もれるからです。つまり、コードを修正できた場合でも、それをリポジトリにはコミットしないことがあります。その場合には、コードの修正案をバグ報告票に貼ってとっておき、このバグを「このバージョンでは修正しない」という形で解決します。次のバージョンの開発が始まったら、これをなるべく早い段階でコミットします。

# トリアージ基準の変化

トリアージは開発の初期段階から意識して行うべき作業ですが、特に開発の末期段階ではトリアージが重要です。つまり、トリアージの基準は時間とともに変化します。図5 - 14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 - 15 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。つまり、開発の初期段階では、あまり深刻度と優先度を区別する必要はありません。単純に、深刻度が高いバグから順に治していけば良いのです。

しかし、開発の末期になるとリソースが残り少ないため、もしリグレッションが発生しても、これを発見して治すことができないかもしれません。そこで、深刻度の低いバグは修正をあきらめることになります。ただし、やはり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。

では、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のでしょうか？いいえ、違います。図5 - 16を見てください。これは、X軸は図5 - 15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 - 16 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。リグレッションテストをやり直す必要がないので、テストへのインパクトもほとんどありません。このようなバグは、リリース前でも治してしまって構わないのです。ただし、いつもより慎重に作業する必要があります。次に何かあっても、このバージョンの製品を修正する機会はもうないからです。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します。

悩ましいのは、深刻度の高いバグの修正は、リグレッションリスクも大きくなりがちであることです。そのようなバグはなるべく早い時期に発見し、ゆとりをもって修正しておかないと、つらいトリアージ作業が待っています。

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。特に、リリース前のリグレッションテストの実施前と後では、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。

# 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、といった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。

このほか、たくさんのPCがコンピュータウイルスに感染したとき、ウイルスを除去できるPCと除去できないPCを選別したり、ソフトウェア開発の初期段階でユーザーからの要求を取捨選択するときなども、トリアージという語を使うことがあります。

人命を選別するとはけしからん、と医療のトリアージを批判する人もいます。また、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ない、という人もいます。しかし、本来トリアージという語には「選別」という意味しかなく、それ以上でも以下でもありません。トリアージは、ソフトウェアの世界でも非常に役立つ考え方です。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。

ひとつ覚えておいてほしいのは、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。これは、ソフトウェアのトリアージが医療のトリアージと全く異なる点です。リグレッションを恐れること、修正しない勇気をもつことを忘れないでください。また、もし修正すると判断したのなら、リリースのスケジュールを遅らせる勇気が必要となる場合があることも覚えておいてください。

図5 - 17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」ことに決めたバグも含まれることに注意しましょう。



図5 - 17 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 - 18 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に感じたら、各ビルドで実施しているテストケースを確認したり、バグバッシュ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、週あたりの新規に見つかるバグ数と、同じ期間で開発チームが解決 (修正) できるバグ数の差が重要です。治せるバグの数よりも見つかるバグの方が多ければ、オープンなバグの数は増え続けていくからです。週あたりの新規に見つかるバグ数が減っていき、これを同じ期間で修正できるバグ数が上回るようになれば、たとえ現在オープンなバグの絶対数が多くても、それは収束に向かいます。こうなれば、バグが収束し、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 - 19 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 - 19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 - 20 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。グラフの右端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。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。別の未解決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、バグが発生しなかった状態」のように定義することもあります。

2回目のZBBや3回目のZBBを経験し、各ZBBの間隔が短くなってくれば、リリース可能な時期が迫っていると考えられます。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  解決されたバグ: 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、バグ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。この、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。

このほかにも、有用な統計情報をグラフにして表示できるBTSがあります。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、これらの機能を活用して、ビルドの健康を管理してください。

# 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が有効です。  バグバッシュ (Bug Bash) とは、そのよう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、チーム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、つまらないバグ出しをみんなで楽しくさっさと済ませてしまおうというアイディアなのです。ただし、会議室に多くのテスト環境を構築するのが面倒であれば、各自のデスクで粛々とテストする、といったスタイルのバグバッシュもありです。最近バグがあまり出てこないとか、リリースが近づいているとか、そのような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ます。もちろん、テストチーム以外の人にもテストを手伝ってもらうわけですから、テストリードが勝手にバグバッシュの開催を宣言しても、皆の協力を得るのは難しいでしょう。人的リソースに関して権限のある人に許可をもらってチームに開催を告知し、メンバーには積極的な参加を促す必要があります。参加しようとする意欲の持てる、楽しいイベントになるように心を配ることです。例えば、軽食を用意するとか、各人の成績 (バグを発見した数) を発表して優秀者には景品を出す、などです。  一般に、バグバッシュはテストケースを準備しないで実施しますが、効率よくバグ出しをするために、おおまかにテストの分担を決めて実施することもあります。もちろん、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。   |  | | --- | | To: 開発プロジェクトチーム  From: テストリード  Date: 2010/12/16  Subject: Bug Bash  みなさま  以下の要領でバグバッシュやります！  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。  …  プロジェクトリードとデブリードには  許可を得ています。  ご協力、よろしくお願いします。 | |

# 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優先順位づけられ、各担当者にアサインされます。このため、バグの修正以外の作業も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、自分のタスクを自分で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[[11]](#footnote-11)。つまり、各ハードウェアに管理票 を仮想的に貼り付けて、これを今誰が使っているのか追跡するのです。BTSは、管理票の変更履歴を自動的に記録してくれますし、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、エクセルでPCを管理するよりもずっと扱いやすくなります。エクセルでバグを追跡するよりもBTSを使った方が便利なのと同じ理屈ですね。BTSは、ソフトウェア開発の現場には、非常に活用範囲が広いのです。

# BTSの二重化

さて、このように大変便利なBTSですが、一般的にはユーザーには開示しないことがほとんどです。詳細すぎる中間成果物をユーザーが閲覧しても、開発業務に混乱を招いてしまうだけだからです。

しかし、ユーザーからの要求やクレームなどに対応するとき、議事録に書きとめてもうまくいきません。議事録はすぐに過去のメールに埋もれてしまい、最新の状態を追跡できるようにはなっていないからです。そこで、ユーザーからの要求を追跡するためのBTS (ITS) を別に構築することがあります。



図5 - 21 BTSの二重化

このようなユーザー対応の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。また、このチケットを有償とすることもあります。この場合、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になります。一般に、ユーザー対応のためのITS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経てクローズします。

また、ソフトウェア開発に特化した高機能なBTSを指してもITSということがあります。ITSにはバグ報告票以外にも、新機能の提案やタスクリストなどが、課題 (Issue) としてファイルされるからです。また、その言葉のわかりやすさから、ソフトウェア開発に特化した最近のオープンな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。

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も多いので、ユーザーにITSに直接起票したり閲覧してもらうこともできるでしょう。また、図5 - 21に示したITSはエクセルシートで実現し、これをユーザーと共有することもあります。ぜひ、みなさんの組織に適切な方法を探してみてください。

# 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るようです。



図5 - 23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[[12]](#footnote-12)。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 - 24 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。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。課題報告票を滞りなくチーム内に流通させることが、非常に重要であることがおわかり頂けるでしょうか。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 - 24を見て、課題報告票の総枚数はコントロールできないが、人的リソースの総人数はコントロールできるのではないかと思った読者もいるでしょう。確かに、より少ない人数で開発できれば、そのコストを抑えられます。しかし、プロジェクトの途中でプロジェクトメンバーの人数を増減させることは困難です。実際には人的リソースが足りなくなり、開発人数を増やしたくなることが多いものです。しかし、ほとんどの場合、これはうまくいきません。新しいメンバーをトレーニングし、キャッチアップしてもらうコストが必要になるからです。赤ちゃんが産まれるまでには10ヶ月かかるのであって、男の人が何人いても役には立ちません。同様に、開発スケジュールを圧縮するために人員を途中で追加投入しても役に立たないことが多いのです。プロジェクトの途中で人的リソースの人数を調整することはできないと考えた方が良いでしょう。





図5 - 25 かんばんの運用ルール





図5 - 26 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ファイルされたときには、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若干違うことが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) していくのが現実的な運用であろうと思います。

# 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)というアプローチが認知されつつあります。チケット駆動では、ITSの活用方法や、チケットの粒度などについてより深い洞察がなされています。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/>

## チケット駆動開発

Redmineによるタスクマネジメント実践技法

<http://www.amazon.co.jp/dp/4798121622>

## Tracer Bullet Development

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>

## 2001年宇宙の旅

Arthur C. Clark

<http://www.amazon.co.jp/dp/415011000X>

## 冷たい方程式

Tom Godwin

<http://www.amazon.co.jp/dp/4150103801>

1. アサイン (Assign) とは、割り当てるという意味です。担当者のほか、プログラムでメモリを割り当てるときにもアサインという語を使います。 [↑](#footnote-ref-1)
2. もちろん、最後のは冗談です。でも、あまり笑えません。 [↑](#footnote-ref-2)
3. 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ことを一件一葉といいます。 [↑](#footnote-ref-3)
4. あるいは、当該のバグを「修正しない」ことにした理由を伝えることができるかもしれません。 [↑](#footnote-ref-4)
5. 私の手元にある複数の辞書には、severityの訳語として「重症度」「過酷度」「発病度」というのが見つかりますが、「重要度」はありません。 [↑](#footnote-ref-5)
6. コミットコメントの書式については、★表3-4を参照してください。 [↑](#footnote-ref-6)
7. 逆に、開発チームも知らなかったバグは、たいてい許してくれません。 [↑](#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. ハードウェア資産番号は、PCに物理的に貼っておきます。 [↑](#footnote-ref-11)
12. 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。この電子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-12)