ASTER 通信カラオケシステム

テスト設計報告書

文書識別番号：TDD-01

Ver. 0.8

2018 **年** 2 **月** 1 **日**

改訂履歴

改訂日	バージョン	改訂者	改訂内容
2017/07/30	0.1	えのき	前提条件を整理
2017/09/04	0.2	えのき	背景とコミュニティの課題を記述
2017/11/18	0.3	えのき	章単位で何を書くかを書く
2017/11/19	0.4	えのき	詳細化して全面的に記述
2017/11/19	0.5	えのき	抜けていたところを記述。レビュー反映
2017/12/25	0.6	えのき	フォールト原因やテスト詳細設計などアップデート
2018/1/30	0.7	えのき	レビュー指摘事項のアップデート
2018/2/1	0.8	えのき	フォールトリスト抜けの検討事項を追記

1. はじめに

1.1 本ドキュメントの目的

本書は、テスト設計方法ならびにテスト設計の妥当性を確認するため、テスト設計コンテスト’18 における「ASTER 通信カラオケシステム」に対するテスト設計活動の概要を示すと共に、テスト設計活動の成果物（成果物２:テスト設計に関わる成果物一式）について説明する。

テスト設計活動とは、以下の活動を指す。

活動	内容
テスト要求分析	テストで何がやりたいかを明らかにする。
テストアーキテクチャ設計	ステークホルダーのテスト要求からテスト戦略、テスト設計方針を決める。
テスト詳細設計	テスト観点をボトムビューポイントまで詳細化し、テストケースを生成する。

1.2 明示的テストベース

今回のテスト設計活動に対して明示的に与えられたテストベースは以下のドキュメントである。

ID	文書名	備考
1	ASTER カラオケシステム要件定義書_Ver2.pdf	最新版(対象テストベース)
2	ASTER カラオケシステム用語集_Ver1_2.pdf	最新版(対象テストベース)
3	ASTER カラオケシステム QA 管理表.pdf	最新版(対象テストベース)
4	ASTER カラオケシステム補足資料_V1_1.pdf	最新版(対象テストベース)
5	ASTER カラオケシステム要件定義書_V1_1.pdf	
6	ASTER カラオケシステム用語集_V1_1.pdf	

2. 背景と前提

ビジネスがモノからコトやサービスに移行したと言われて久しく、ソフトウェアがビジネスの成否を規定するケースが多い。一からソフトウェアを作るのではなく、オープンソース・ソフトウェアや他社のクラウドサービス等を利用することで、ビジネスの速度が年々加速している。オープンソース・ソフトウェアをいかに使いこなし、品質を確保するかのノウハウは、イノベーションを起こしたい多くの組織においてますます重要になってきている。

ところが、オープンソース・コミュニティにおいて、いかに貢献者を獲得し、品質を確保していくかのノウハウは発展途上である。また、企業においてもオープンソースをいかに活用し、貢献していくかについて、ノウハウはあまり共有されていない。

そこで、てすにゃん v3 では、オープンソースプロジェクトに置ける品質確保について検討する。架空のオープンソースカラオケソフトウェア LibreKaraoke を想定し、そのコミュニティ内でのテスト設計を行うことで、オープンソースでの品質と貢献のあり方の一例を示す。

2.1 LibreKaraoke 製品とコミュニティ概要

架空のオープンソース LibreKaraoke は、Linux 上で動作するのカラオケソフトウェアである。元々は x86 向けの Linux 上で動作するように開発されていたが、Android 向けも開発されるようになった。LibreKaraoke を利用したカラオケ専用マシンを作るベンダが複数出現し、クラウドでのカラオケサービスを展開するベンチャーも登場している。

2.2 歴史

2003 年ごろから、カラオケシステムが普及していなかった国・地域に住む開発者によって趣味として開発が開始された。C++で記述され、プラグイン機構も備えている。ボランティアベースのコミュニティが形成され、最低限の機能からスタートして、順次機能が追加されて、高機能なカラオケソフトに成長した。

特殊な音声用のチップがなくてもカラオケ機器を開発できるようになってきたことから、後発カラオケ機器ベンチャーなどはこれをベースにカラオケマシンを開発・販売するエコシステムが育った。現在このソフトウェアを組み込んだ主なカラオケ機器メーカーは 3 社、売上は合計約 1 億ドルである。

レガシーなアーキテクチャやコードをリファクタリングを進め、コア部分を切り出して API の整備を行うことで、パソコンとカラオケ機器、さらには Android（開発途上）などの他のアーキテクチャへの対応を進めてきた。

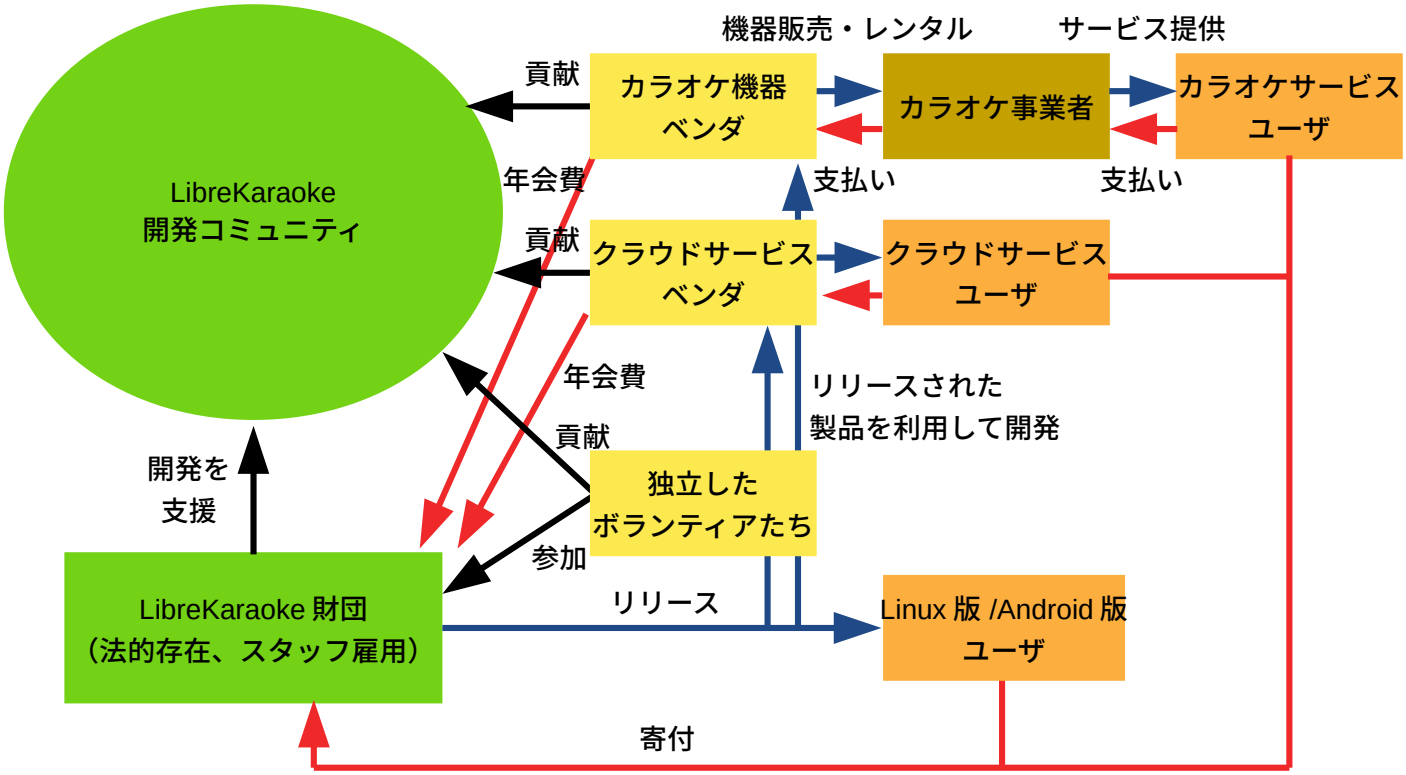


図 1: LibreKaraoke エコシステム（生態系）

2.2.1 開発体制

コミュニティベースで開発されており、コミュニティを支える非営利法人である LibreKaraoke 財団を設立している。フルタイムのメンバとしては、財団に直接雇用された貢献者が 7 名、カラオケ機器ベンダが 10 名、クラウドカラオケサービスベンダに 4 名いる[i]。また、独立したフルタイムでない多くの貢献者によるコードコミットも約 1/3 を占めている。

2.2.2 開発プロセス

ソースコード管理を Git で管理しており、オープンで透明性の高い文化を目指しており、貢献を広く求めている。

開発フローは、開発の提案内容を ITS（Issue Tracking System）の Issue として起票し、開発メーリングリストで議論する。大きな影響がある場合は技術委員会でコンセンサスを取る必要がある。小さな変更の場合はいきなりコードを書く場合や、Issue のみ起票するケースもある。動作するソフトウェアがないと説明しずらい場合には、まれに先に開発しておいて、後で議論やコンセンサスをとるケースもある。ただし、開発したコードが無駄になる可能性は高いため、通常は行われない。

開発者がソースコードを push し、コミット権限のある開発者がソースコードレビューを行い、コメントを通じて修正アドバイスをしたり、取り込むかの決定をする。ソースコードレビューを通じて新しい開発者の教育的な意味を果たすこともあり、はじめてコミットされた場合にはコードの書き方のルールなども含めて丁寧にレビューすることが多い。なお、開発した本人がコミット権限があったとしても、自分でレビューしてセルフコミットすることはポリシー上認められていない。

レビューで承認された場合は自動的に Master ブランチにマージされて、ビルドボットによりビルドされる。また、テスト用環境に自動的にデプロイされてビルド時用に用意されている自動テストが実

行される。リリースはリリースエンジニアがタグをうち、ビルドされたイメージもコマンドで半自動的にリリースされる。[ii]

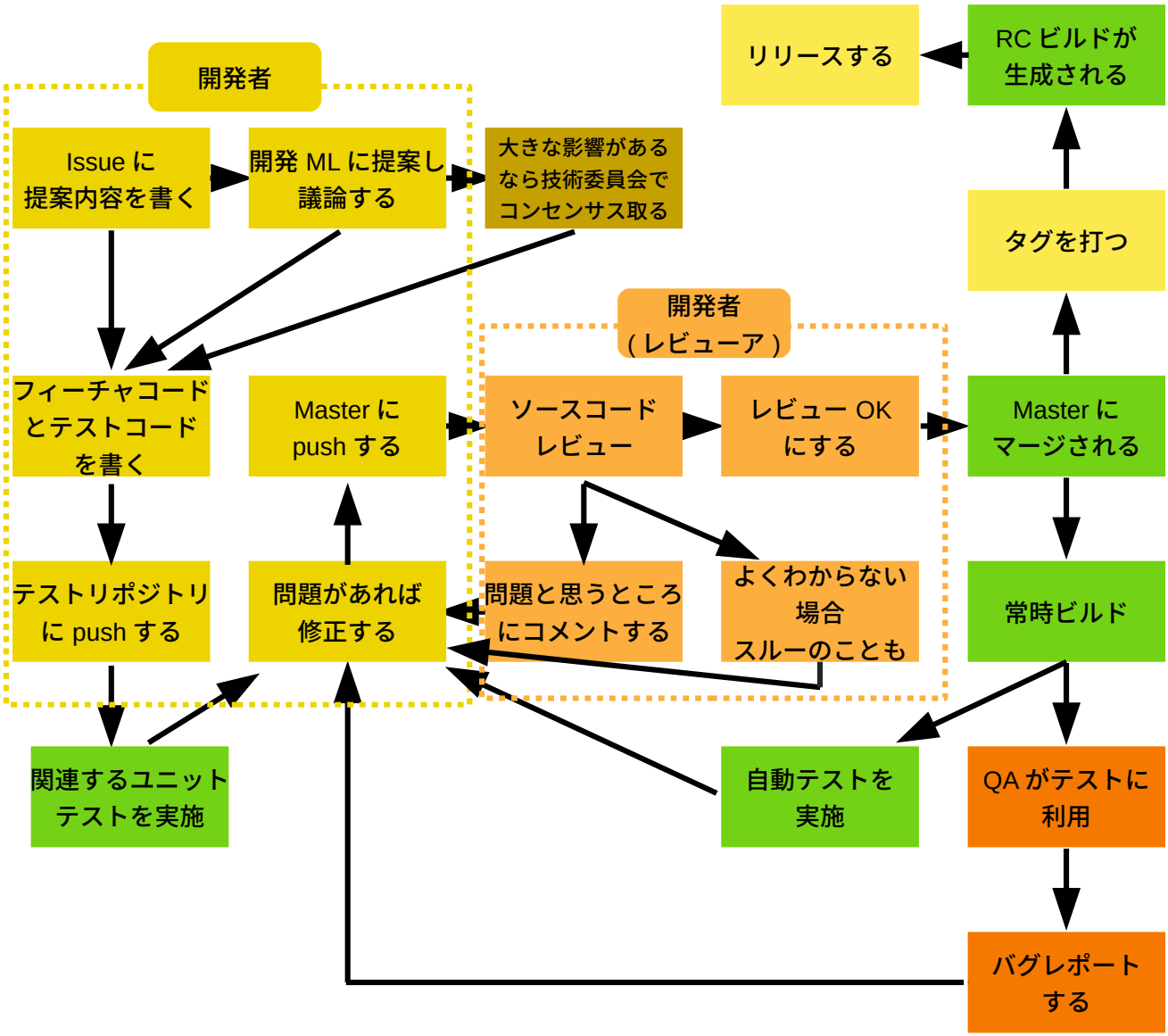


図 2: 開発プロセス

2.2.3 リリース戦略

素早いフィードバックをもらうことが重要であり、対象ユーザによって許容できるリスクが違うため、リリース頻度を変えている。安定版は 6 ヶ月に 1 回リリースを行っており、リリースに向けて取りまとめを行うコストを下げたり、ユーザー側が計画を立てやすくするためにも Ubuntu や Gnome、LibreOffice などのように時期が来ればリリースするタイムベース・リリースを採用している。開発版は積極的なユーザ向けであり、2 週間に 1 回リリースしている。小規模なアップデートが自動的に繰り返される、ローリング・リリースである。また最新状態をテストしたい QA メンバ向けに自動アップデート機能がついたデイリービルドを提供している。

安定版に対しては、コミュニティからクリティカルバグとセキュリティ修正のアップデートパッチを1年間提供している。それよりも長く利用する場合は、各自で Master からセキュリティの修正などをチェリーピックしてきてビルドして対応する。例えば、Linux ディストリビューションの Debian での LibreKaraoke パッケージなどはそのようにして、およそ3年くらいは[iii]セキュリティ修正を出している。

2.3 てすにゃんの立ち位置

LibreKaraoke コミュニティによって設立された財団に雇用された QA と QA インフラを担当するエンジニア2名の立場とする。コミュニティにおける QA 活動のコーディネート、テスト設計やテスト実装、テストインフラの構築と運用などを行っている。

カラオケ機器ベンダやクラウドサービスベンダの QA 活動は、ほとんどコミュニティからは見えていないが、一部の活動はコミュニティの中で行われている。ベンダの QA 活動のコミュニティ側への誘導やベンダ QA とのコミュニケーションも重要な任務の1つである。

3. テスト設計コンセプト

品質における課題は多様である。今回、緊急の課題として「プロダクトのリリースクリティカルバグを許容範囲まで減らす」ことを目指す。具体的な目標としては、安定版リリース時までにリリースクリティカルバグを発見除去する比率を現在の約 85%から、1年以内に 97%以上に高めることを目指す。また、最新版については現在の約 60%から1年以内に 90%以上に高めることを目指す。なおこれは、コミュニティとしての目標値であり、外部に対してのコミットメントではない。

長期的にはソースコードの品質をあげて、バグを混入しづらいようにすべきである。現在はリリース後のバグ修正に開発上の無駄なコストがかかっており、早い段階でのクリティカルバグの除去によって開発速度を向上させること、ユーザーにとっても安心して利用できる状態を作り出すことを狙う。

3.1 開発コミュニティでの品質課題

いくつかの問題が認識されており、特に議論になっている4つを取り上げる。今回のスコープは、これの１番目の課題に対応するものである。

1. リリースされた製品にリリース基準を満たしていない、重大なバグが残っていることが多い。
2. ソフトウェアのアーキテクチャとソースコードの内部品質の問題があり、コードを修正するとバグが発生しやすく、リグレッションも起こりやすい状況である。
3. バグレポートなどのフィードバックを十分に捌ききれていない。一方でユーザがバグを発見しても報告する手間が大きく、そのまま報告されないケースが多い。
4. 企業側とコミュニティとの QA 活動の連携が不十分であり、どこをそれぞれがテストしているかが見えてない状況である。

3.2 課題に対する解決策

1. リリース時に重大なバグを残したままリリースすることを減らすため、リスクを洗い出し、今回のテスト設計からテストを実施する

2. ソフトウェアのアーキテクチャとソースコードのリファクタリングを進めるなどが考えられるがリスクも伴う。また、特に優秀な開発者でなければ実現できないため、開発リソースをどう確保するかが課題である。
3. バグトリアージを行いバグレポートをさばいていく必要がある。ボランティアでも再現確認は取り組みやすいために、誰でも簡単に取り組めるようナレッジとインフラを整備する、ボランティア向けのキャンペーンを実施するなどの作戦がありえる。
4. 企業側にコミュニティで QA 活動を共有するように働きかける。合意できた場合、テストケースやステータスを共有する仕組みを作る。

4. テスト要求分析

テスト要求分析では、リリースクリティカルバグをリリースまでに削減するという課題に対して、どのリスクに対して対応するかを明らかにする。ここで行う３つの活動を説明する。

- 品質のアクティビティを概観する：現状、誰がどのような品質に関わる活動をしているかを整理するため、図を作成する。必要に応じてヒアリングなども行う。
- 製品リスクを整理する：起こって欲しくないこと（フォールト）とその原因をツリー図として書き出す。
- ソースコードのリスクをチェックする：オープンソースのバグ予測ツール「Bugspots」を使ってソースコードでのリスク値を出す。

4.1 品質のアクティビティを概観する

現在どのような活動を行っているかを確認する必要がある。一覧図にまとめると分かりやすい。

開発者はソースコードレビューを基本としつつ、ユニットテストを開発者の判断で必要と判断したケースでは書いている。また、特定の開発者のタスクとして、ファジングツールと静的解析ツールで問題を洗い出してコードの修正を行っている。

ユーザーは、開発版や安定版を通常で利用したり、最新版や開発版での気になる機能のチェック・新機能のチェック、バグレポートなどを行っている。

QA チーム（財団の QA エンジニア、ベンダのエンジニア、独立のボランティア）としてはエンドツーエンドの自動テストをごく一部と、リグレッションテストの手動テストを一部のみ、新機能や修正箇所のテストの一部、バグレポートとバグトリアージを行っている。フィーチャーテストのまとまったテストケースセットは持っていない。

カラオケ機器ベンダーやクラウドサービスベンダー内部はコミュニティからは見えないものの、バグレポートなどのフィードバックやコミュニティに参加しているベンダ所属のエンジニアからのヒアリングから、リグレッションテストやフィーチャーのテスト、パフォーマンステスト、UX テスト、システムテストなどを行っているようである。

今回の取り組みによって、「フォールトテスト」を新たに追加する。「エンドツーエンドの自動テスト」「新機能、修正箇所のテスト」「リグレッションテスト」のうち「フォールトテスト」でカバーできるものはそちらに移行させる。

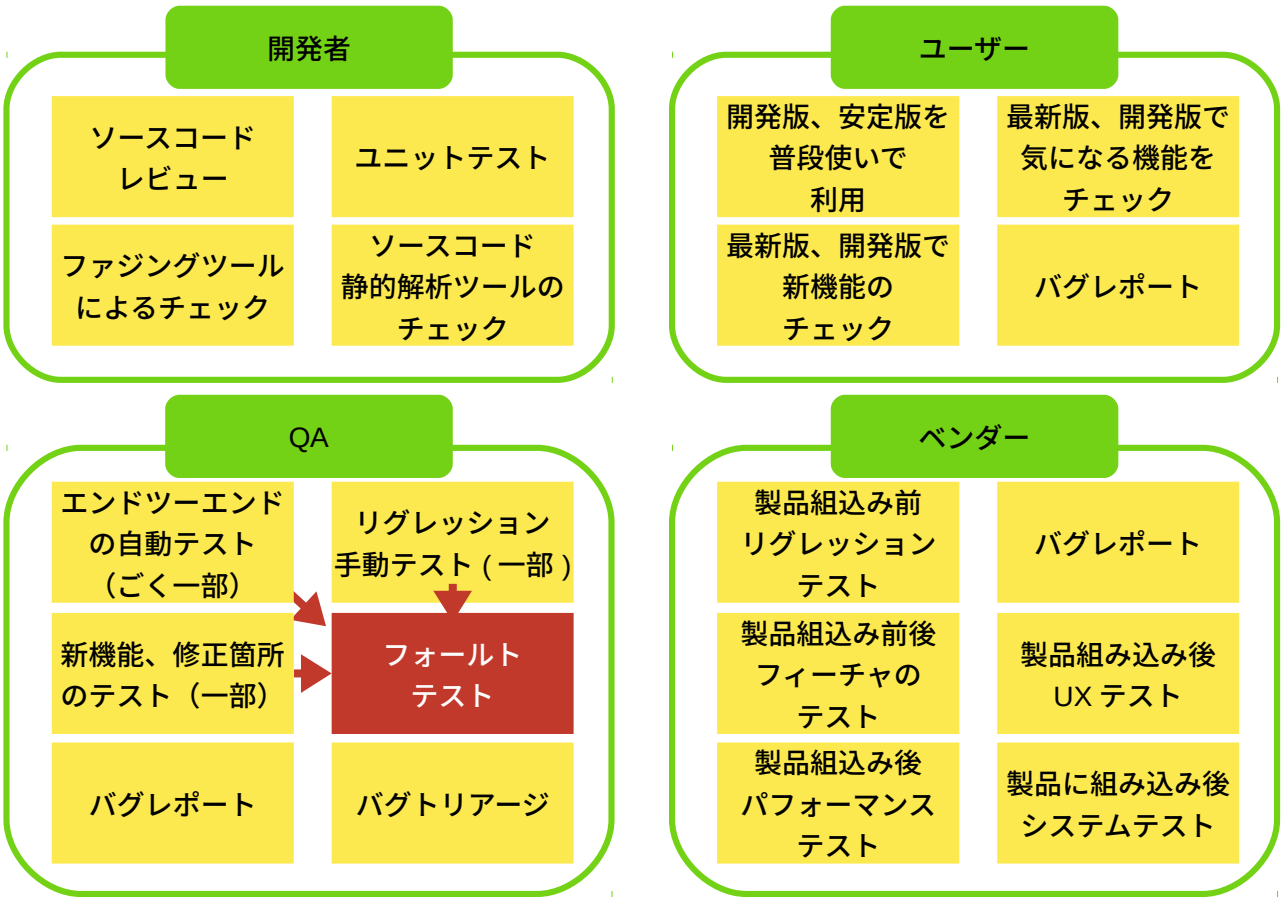


図 3: 品質のアクティビティ一覧（従来に対してフォールトテストを追加する）

4.2 製品リスクを整理する

製品の目的に照らしたリスクや、過去不具合に出ていないリスクに対処するためには、トップダウンでリスクを考える必要がある。起こって欲しくないこと（フォールト）を洗い出して、グルーピングして整理する。それらに対して原因を分解し、フォールトツリー図（FT 図）のような形式で整理する。原因を潰していくことで、起こって欲しくないことの発生を減らすのが狙いである。

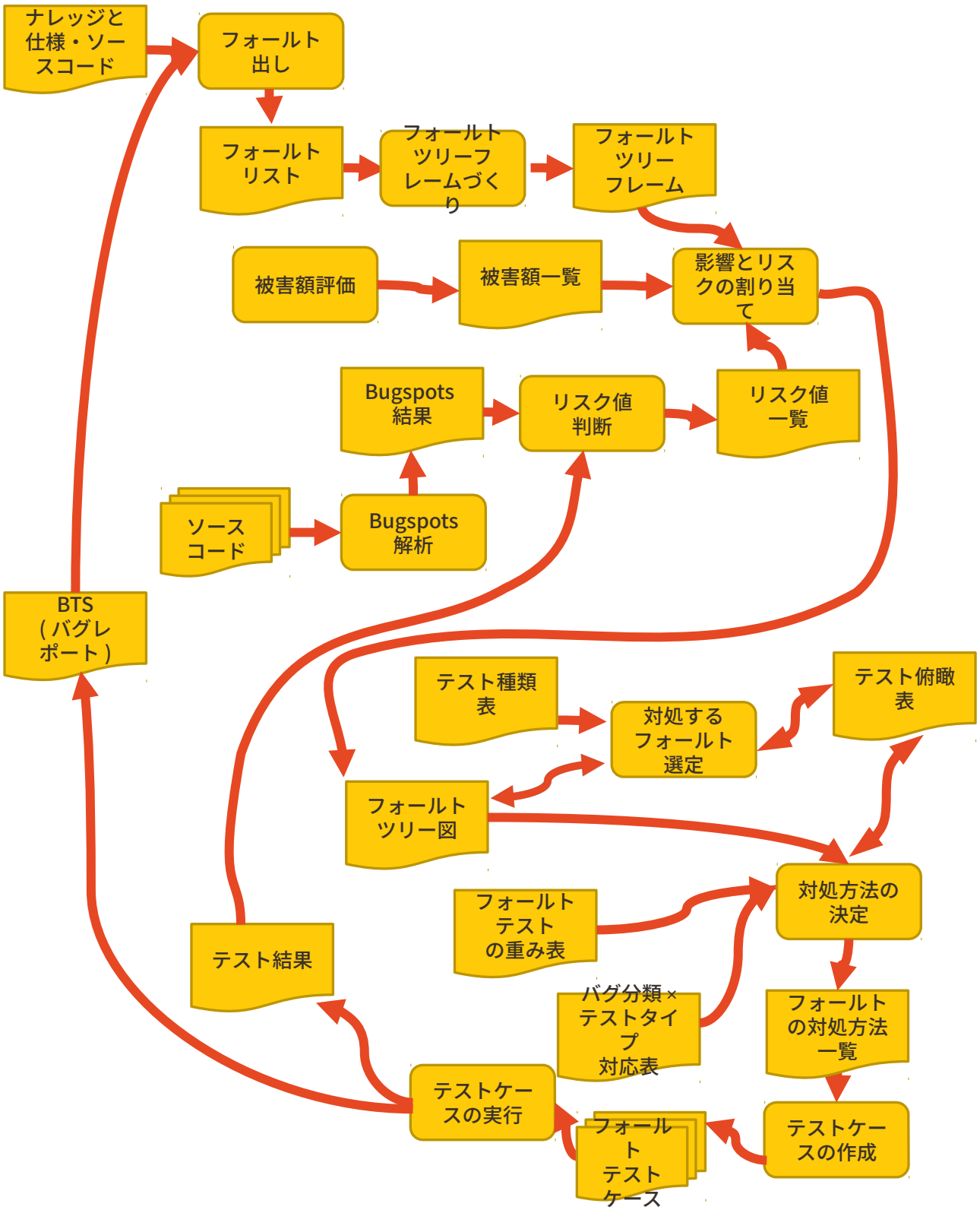

ナレッジと仕様・ソースコード
フォールト出し
フォールトリスト
フォールトツリーフレームづくり
フォールトツリーフレーム
被害額評価
被害額一覧
影響とリスクの割り当て
Bugspots結果
リスク値判断
リスク値一覧
ソースコード
Bugspots解析
BTS(バグレポート)
テスト種類表
対処するフォールト選定
テスト俯瞰表
フォールトツリー図
対処方法の決定
フォールトテストの重み表
テスト結果
バグ分類×テストタイプ対応表
フォールトの対処方法一覧
テストケースの実行
テストケースの作成
フォールトテストケース

図 4: テストの全体図

4.2.1 起こって欲しくないこと（フォールト）の導出方法

身体的リスク、金銭的リスク（課金）、うまく使えないリスク、物理的破壊リスク、社会的要請の 5 つの観点から起こって欲しくないことを検討する。このように分類を行う理由は、ユーザーにテストを任せるものと、QA や開発側が行うものに分けるためである。テストの人的リソースが少ない中で効率的にリスクを削減するためには、ユーザーによるテストの力を最大限に活かす必要がある。

「身体的リスク」は必ず回避したいものであり、大きな「金銭的リスク（課金）」も対処する必要があるが、どちらもユーザーがテストするにはふさわしくないものである。一方「うまく使えないリスク」はユーザーが利用する中で発見することが容易であり、ユーザーテストに任せることが可能である。また、それらに収まらないものがあり、「物理的破壊リスク」「社会的要請」として分類した。

５つの観点について数名で半日程度かけたワークショップを開催し一気に洗い出す。考えついたものを各メンバがそれぞれ付箋に出す。書きだした出したものを重複などを整理し、どうしても起こって欲しくないことと、出来れば起こって欲しくないことに分類して、前者のみを採用する。最終的に、ワークショップ参加メンバの合意によって導出する。

フォールトとなるナレッジは、プロジェクトの過去不具合（BTS）、類似プロダクトの過去不具合、別分野での起こりがちなリスク、ドメインに起因する制約事項、社会的な要請など、形式知となっているものと、分析に参加するメンバーの暗黙知からなる。形式知については、ワークショップ前に整理しておけるとよい。

フォールトの導出精度はワークショップ参加メンバーに依存する。製品の理解度が高い QA エンジニアや開発者、また多様な立場のメンバが確保できるとよい。なお、完璧なものを考え出すことは困難であり、今後もアップデートしていくため、この時点ではほどほど出せればよいとする。

- 身体的リスク：利用する中でユーザーが身体的な損害が起こりえるもの。例としては、「音量コントロールがうまくできず、大音量で鼓膜が破れる」などである。
- うまく使えないリスク：ユーザーがカラオケを利用する中で期待通りに動作しないことである。例としては「音が流れない」「採点されない」などである。フィーチャーやパフォーマンス、UX に関わる部分であり、大量に出すことが可能である。しかし、ユーザーによるテストに任せるので、ここを出すのに夢中になって時間を使い過ぎないようにする。
- 金銭的リスク（課金）：ユーザだけでなく、カラオケ事業者の被害についても考える。例としては「課金の計算が正しく行われない」などである。
- 社会的要請：社会を円滑に運営するために法律やその他のルール、ルール化されてないものの強く要請されるケースもある。例としては「ユーザのデータが外に漏れる」ことなどである
- 物理的破壊リスク：ハードウェアの破壊とそれに伴うリスク。例としては「バックアップデータが壊れる」などである。

身体的リスク	不快な音波
	大音量
	気分が悪くなる映像が流れる
	チカチカ(光の点滅)
うまく使えないリスク	ハウリング
	音が出ない
	映像が出ない
	字幕がずれる
	途中で止まる
	テンポが変わる
	字幕が切れる
	となりの音が割り込む
	起動しない
金銭的リスク(課金)	課金ができない
	課金の計算が正しくない
	課金データが改ざんされる
社会的要請	クラッキングされる
	踏み台にされる
	操作データが外部にもれる
	楽曲データがとられる
物理的破壊リスク	ハードウェアが壊れる
	データ破壊により営業が不可能になる
	サーバーのデータを破壊する
	大量にデータをDL

図 5: 起こってほしくないこと（フォールト）の一覧

4.2.2 HAZOP を利用したフォールトの追加洗い出し（この部分は後フェーズの成果物に未反映）

フォールトを全部出しきれていないのではないか、との議論がコミュニティ内であった。今回フォールト抽出ワークショップ・メンバのドメイン知識などに不安もあったこともあり、QA チームとしても自信を持つことができなかったことから、追加で HAZOP を利用したフォールトの洗い出しを行った。

HAZOP を選んだ理由は、同じメンバで同じように検討するだけでは新たな抽出が難しいと想定され、強制的に異なる視点で検討できる必要があったからである。

これにより、フォールトを 11 追加で発見し、15 個から 26 個に増加した。今後の作業として、フォールトのレベルにばらつきがあり、似たようなものをまとめる必要がある。追加で洗い出した分については、フォールト一覧の更新以降の作業はまだ着手しておらず、今後実施する。

洗い出しの方法としては、まず、要件定義書の第 1 章や第 2 章から本当の要求と考えられるものと、コミュニティの要請から、最上位の要求リストを作成する。この要求リストについて、コミュニティ内で合意形成を行う。

ステークホルダー	要求	
ユーザ	気持ちよく歌える	いい音で歌える
		きれいな映像が流れる
		スムーズに使える
	歌いたい曲を歌える	-
オーナ	お客さんが利用してくれ、儲かる	課金ができる
		ハードウエアが継続的に使える
		データが継続的に使える
		運用コストが高すぎない
ベンダ（サプライヤ）	ハードが売れる	-
	財産（曲、コンテンツ）を守れる	-
コミュニティ	自由な環境を作りたい	-
	社会に悪影響を与えたくない	セキュリティのリスクがない

図 6: 要求リスト

続いて、要求だけでは大きすぎるので、要求を分割したものに対して、ガイドワードをかけたマトリクスを作成した。なお、ガイドワードは、河野哲也氏の「ソフトウェア要求仕様における HAZOP を応用したリスク項目設計法」を用いた。

ソフトウェアでのガイドワードとしては、他にも SHARD（Software Hazard Analysis and Resolution in Design）などもあるが、ガイドワードが抽象的であり、使うことが困難であると判断した。そこで、利用された事例を見つけた河野氏のものを採用した。

マトリクスとして出したものに対して、フォールトに該当するものを抽出する。具体的には、数が多いため、この段階でユーザテストなどでカバーする範囲を除いたものをフォールトにすることとした。JIS X 25010：2013 (ISO/IEC 25010：2011)の「3.2 利用時の品質モデル」のリスク回避性である「経済リスク緩和性」「健康・安全リスク緩和性」「環境リスク緩和性」が狙いたいところとほぼ同じであるため、これに該当するものをフォールトとした。

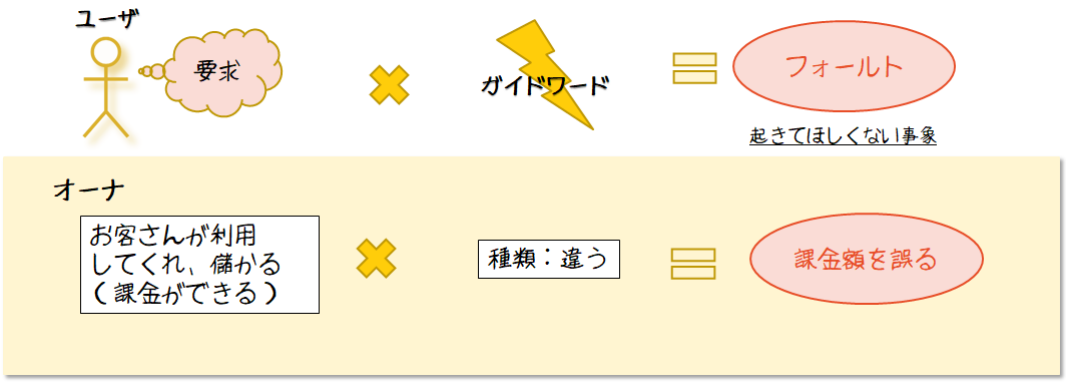


図 7: フォールトの抽出の考え方

そうして作成したものが「てすにゃん V3_成果物 2_16_HAZOP を利用したフォールトの追加抽出_R01.pdf」である。

	Top事象の導出率	7.69%									
			有無	程度		速度		持続時間		範囲	
ステークホルダー	要求		全く〜しない	強く	弱く	急いで	ゆっくり	ずっと	一時的に	余分に	不十分に
ユーザ	気持ちよく歌える	いい音で歌える	ー音がしない	★音が大きすぎる	ー音が小さすぎる	ー曲が早すぎる	ー曲が遅すぎる	-	ー一時的に音が切れる	ー余分な音が鳴る	ー音が少し流れ
		きれいな映像が流れる	ー映像がない	★映像が激しすぎる	-	-	-	-	ー一時的に映像が止まる	ー余分な映像が流れる	ー映像が少し流れ
		スムーズに使える	ー全くスムーズでない	-	-	-	-	-	ー一時的に遅くなる	-	-
	歌いたい曲を歌える	-	ー歌いたい曲が歌えない	-	-	-	ー歌いたい曲が一定期間経過特に追加されろ	-	ー歌いたい曲が一定期間特に歌えなくなる	ー歌いたい曲以上に曲がある	ー歌いたい曲が
オーナ	お客さんが利用してくれ、儲かる	課金ができる	★課金ができない	-	ーあまり利用してくれない	-	ー課金に時間がかかる	-	ー一時的に課金ができない	★余分に課金してしまう	★課金額が足り
		ハードウエアが継続的に使える	★ハードが全く使えない	-	-	-	-	-	★ハードが一時的に使えない	-	-
		データが継続的に使える	★データが全く使えない	-	-	-	-	-	★データが一時的に使えない	-	-
		運用コストが高すぎない	★運用コストが極端に高い（通信費）	-	運用コストが高い	-	-	-	-	-	ー運用コストが
ベンダ（サプライヤ）	ハードが売れる	-	ーハードが売れない	-	ハードがあまり売れない	-	-	-	-	-	-
	財産（曲、コンテンツ）を守れる	-	★曲、コンテンツを守れない	-	★あまり曲、コンテンツを守れない	-	-	-	★一時的に曲、コンテンツを守れないことがある	-	★一部曲、コンツを守れない
コミュニティ	自由な環境を作りたい	-	ー自由な環境を作れない	-	-	-	-	-	ー自由な環境を守れないことがある	-	ー自由さが不十
	社会に悪影響を与えたくない	セキュリティのリスクがない	★セキュリティリスクが特大	-	★セキュリティリスクが大きい	-	-	-	★一時的にセキュリティのリスクが高まる	-	★セキュリティ

図 8: HAZOP を使って作成したフォールト抽出マトリクスのイメージ

4.2.3 フォールトの原因の導出方法

フォールトを出した後、それに対して原因を考える。原因を階層構造にブレークダウンしていく。それをフォールトと原因を紐づけたツリー「フォールトツリーフレーム」として整理する（成果物 2_03_フォールトツリーフレーム）。要因が複数ある場合は、AND や OR で繋ぐ。

フォールトの原因を出す時に、必要に応じてシステム構成図や設計を参照もしくは作成する。フォールトの原因は、どのように作るかに依存する部分が大きく、システムアーキテクチャやその部分の設計を理解する必要があるためである。[iv]

ユーザテストで対応する「うまく使えないリスク」についてはテストケースを作成しないため、原因を導出しない。今後問題が起こった際にユーザーテストだけでは対応が難しいと判断した場合に、原因を導出して、フォールトツリーフレームに追加したうえでテストケースの作成・実施を行う。

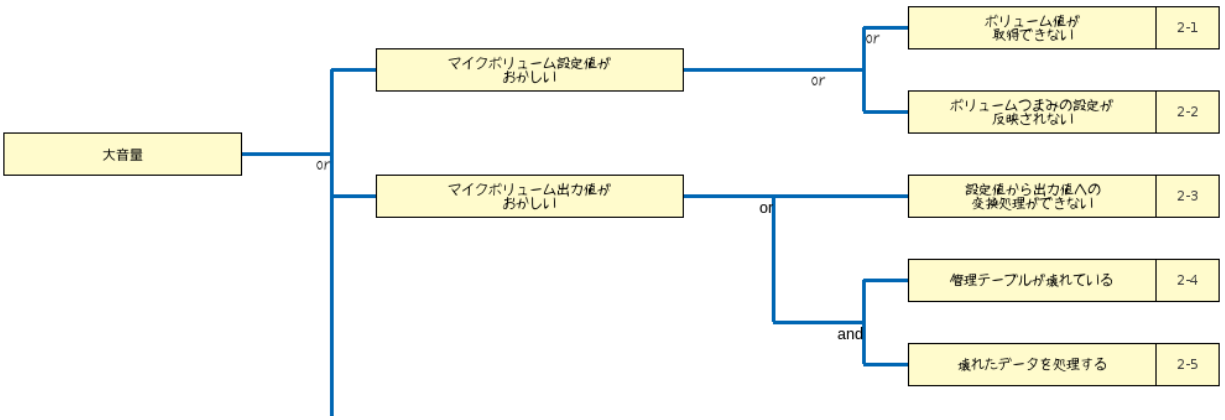


図 9: 1 つのフォールトに対して原因を出した、フォールトツリーフレームの例

主にエンジニアが考えてブレークダウンする。ドメイン知識と経験を要するところであり、QA エンジニアがそれまでに製品を熟知していることが望ましい。また可能であればベテラン開発者のレビューを受ける。

なお、定期的（６ヶ月に１回程度）にバグの報告状況を見ながら、フォールトやその原因を検討し、フォールトツリーフレームを更新する。特に大きな問題があって対応を検討した場合は、定期以外でもアップデートする。

4.3 ソースコードのリスクをチェックする

製品のリスクの側面だけでは、実際にどの程度問題が起こりえるかは分からない。テストに使えるリソースが少ない中で、効率よくリスクを見ていくためには、ソースコード側からもリスクをチェックすると効果が高いと考えられる。

ソースコードのリスク分析には様々なアプローチが研究されているが、今回は、Google が開発したオープンソースのバグ予測ツール「Bugspots」[1]を利用する。理由は、経験に照らしてもそこそこ役に立つ情報が期待できそうとのことと、オープンソースであるためコミュニティで利用しやすいからである。

Bugspots のやっていることはシンプルで、ソースコードがいつバグ修正されたかをみて、問題が起こりやすいホットスポットを見つける。バグが修正されているファイルということは、そのコードに問題がある可能性が高いという判断である。最近の修正ほど値が高くなるようになっている。

具体的には、ソースコードのコミットログから特定文字列（デフォルトでは fix）を含むコミットを抜き出して、ファイル単位でスコアリングされ上位 10%のファイルが抽出される。

プロジェクト初期のバグレポートがほぼない状況では Bugspots は有効ではない。バグレポートとしてフィードバックがかえって来はじめてから利用を開始する。

LibreKaraoke のソースコードに対して定期実行するように設定する。開発者全体で、バグ修正した場合に fix という文字列を含め、それ以外の場合には fix を使わないというコンセンサスを作って周知する。もし、誤ったコミットログを見かけた場合には、開発者に修正を依頼する。

1 https://github.com/igrigorik/bugspots
http://www.publickey1.jp/blog/12/bugspots.html

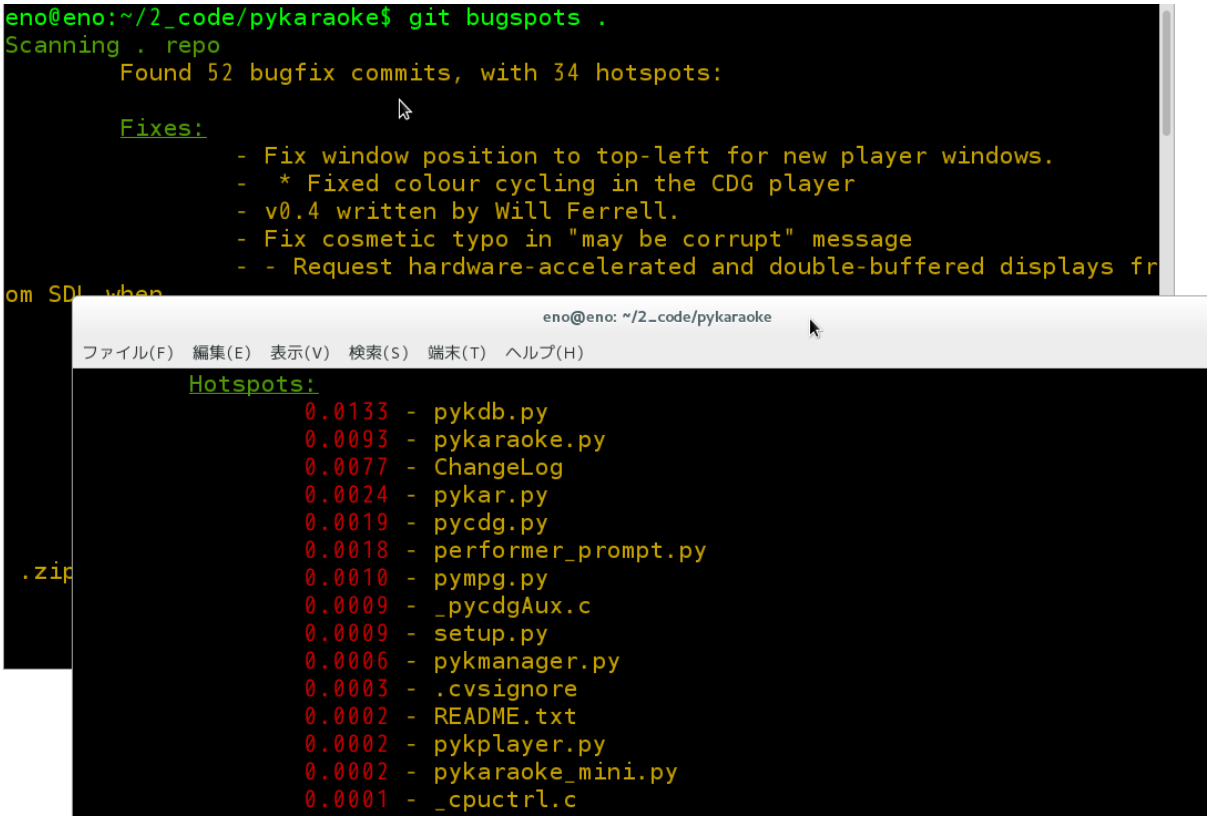


図 10: 実在する OSS の pykaraoke に対して、Bugspots を実行してみた結果

5. テストアーキテクチャ設計

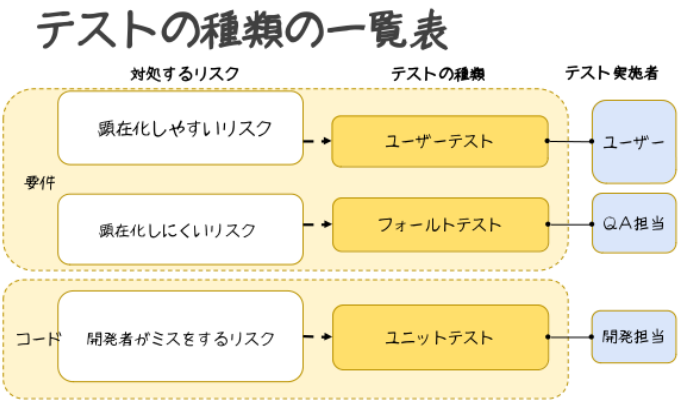
テストアーキテクチャ設計では、テスト要求分析で導出したリスクへの対応枠組みの合意し、個別リスクへの対応を決定する。

- 技術委員会で合意する：誰がどこをどうやって対応していくかの枠組み（テスト設計方針）を合意する
- リスクへの対応を考える（フォールトツリー図）：製品リスクのフォールトツリーフレームと、ソースコードのリスク値、被害評価の一覧を使って、リスクの割り当てを行う。どのリスクを誰がどのようにテストするかを考える

5.1 技術委員会でテスト設計方針を合意する

コミュニティでの品質リスクにどう対応するかは、すべてのコミュニティメンバにとって重要なことであり、多くのメンバの活動に関わることでもある。コミュニティでコンセンサスを形成する必要がある。メーリングリストでドラフトを提案し、フィードバックを求めて議論を行い、最終的には技術委員会の電話会議で議論して合意する。

今回、5 つの起こって欲しくないことへ分けること、「うまく使えないリスク」についてはユーザーによるテストで対応すること、リスクをフォールトツリー図で原因別に整理したこと、ソースコードのリスクとあわせてリスク値を出すこと、テストの重み付け、テストでのリスク低減率の計算方法（成果物 2_12_フォールトテストの重み表）を合意する。



テストの重みづけ		テストタイプ※1 データ	条件	シナリオ	イベント/タイミング	構成
	軽 最低限のテスト	有効/無効	重複結果なし	C0	0-switch	1-wise
	中	＋同値クラス	無効なし	C1	セル網羅	2-wise
	重	＋境界値	完全	C2	1-switch	3-wise

※1 ユーザテストについては、エンドユーザにてテスト実施

リスク値低減度		テストタイプ データ	条件	シナリオ	イベント/タイミング	構成
	軽 最低限のテスト	31.00%	25.00%	56.76%	61.67%	23.57%
	中	71.00%	45.00%	77.35%	83.89%	80.71%
	重	95.00%	95.00%	95.00%	95.00%	95.00%

※テスト実施誤り率:5%と想定した。

図 11: テスト設計方針に含まれる「テストの種類一覧」「重み付け」「リスク低減計算」

実際にはいきなりすべてを提案するのではなく、順番に提案して試していく形になるだろう。最も議論になりうるものは「うまく使えないリスク」はユーザによるテストに任せてしまうことだと考えている。事実上はその状態であっても、明文化することに対してコンセンサスを取ることは困難も予想される。重要なのはユーザーをよりテストに巻き込むことであり、その作戦の一環として提案する。

5.2 リスクへの対応を決定する

5.2.1 フォールトツリーフレームの原因と、ソースコードのリスク値の紐付け

トップダウンのリスクと、ソースコードからのリスクを付きあわせて判断したい。フォールトツリーフレームでの最下層の原因部分のソースコードがどこかを調べる。そして「Bugspots」で出したリスク値が高いファイルとマッピングする。

「Bugspots」で出したリスク値が高いファイルがどのようなことをやっているかを把握して、一覧に書き出す（成果物 2_06_Bugspots 結果）。それとフォールトツリーフレームを見ながらマッピングを行う。ファイル数が非常に多い場合は Bugsopts の値が高いものだけを一覧化して、マッピングを行う。原因のものがファイル一覧にないこともありえるが、その場合はソースコードを探す。

マッピング結果をフォールトツリーフレームに追記し、フォールトツリー図（成果物 2_08_フォールトツリー図）と呼ぶ。

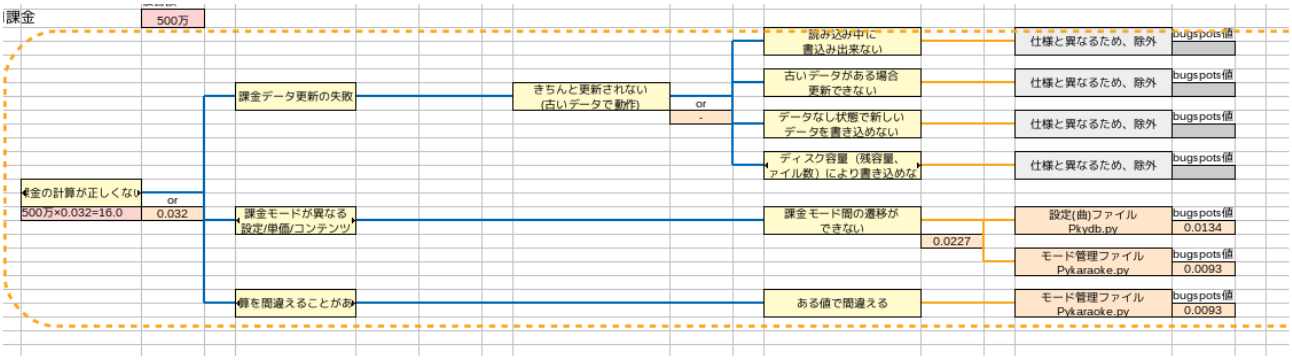


図 12: フォールトツリー図の一部

なお、リスクの見直しなどで 2 回目以降場合には、Bugspots の値が高くても、1 回目にテストを作成などして対応できていれば、Bugspots の値をそのままリスク値として使わず調整する（成果物 2_07_リスク値一覧）。

QA エンジニアだけで理解しきれないケースもある。その部分について詳しい開発者をコミットログから特定し、協力を得ることも有効な手段である。完全にマッピングするには手間がかかりすぎることから、おおよそマッピングできればよい。

5.2.2 対処する/しないや、対処方法を検討する

基準値を設定し、フォールトの被害金額期待値と、フォールトに紐づく個々の原因のリスク値を足したものを掛け算して、基準値を下回るようにする。また、その掛け算の値の順番にテスト等で対処していく。

基準値を下回ることが出来ないケースもあるが、それによってリリースを止めることはない。リリースを延期するのは、リリースクリティカルバグが実際に発見されて修正が間に合わなかった場合のみである。

フォールトの被害金額期待値は、フォールトの 5 つのリスクの種類ごとに一律に決める（成果物 2_04_被害額一覧）。より精度を追求するならば、フォールト 1 つずつに対して被害金額期待値を見積もるべきであるが、その判断は簡単ではないため、簡便な手段を用いる。

基準値は一律 1000 万円と仮決めする。バグの状況などをみて、明らかにおかしい場合は数値を今後調整する。

M x P < B

M：フォールトの被害金額期待値

P：そのフォールトからぶら下がる原因のリスク値を合計したもの

B：基準値 1000 万円

身体的リスク	1000万円
金銭的リスク(課金)	500万円
うまく使えないリスク	今回は対象外
社会的要請	5億円
物理的破壊リスク	10万円

図 13: フォールトの被害金額期待値一覧

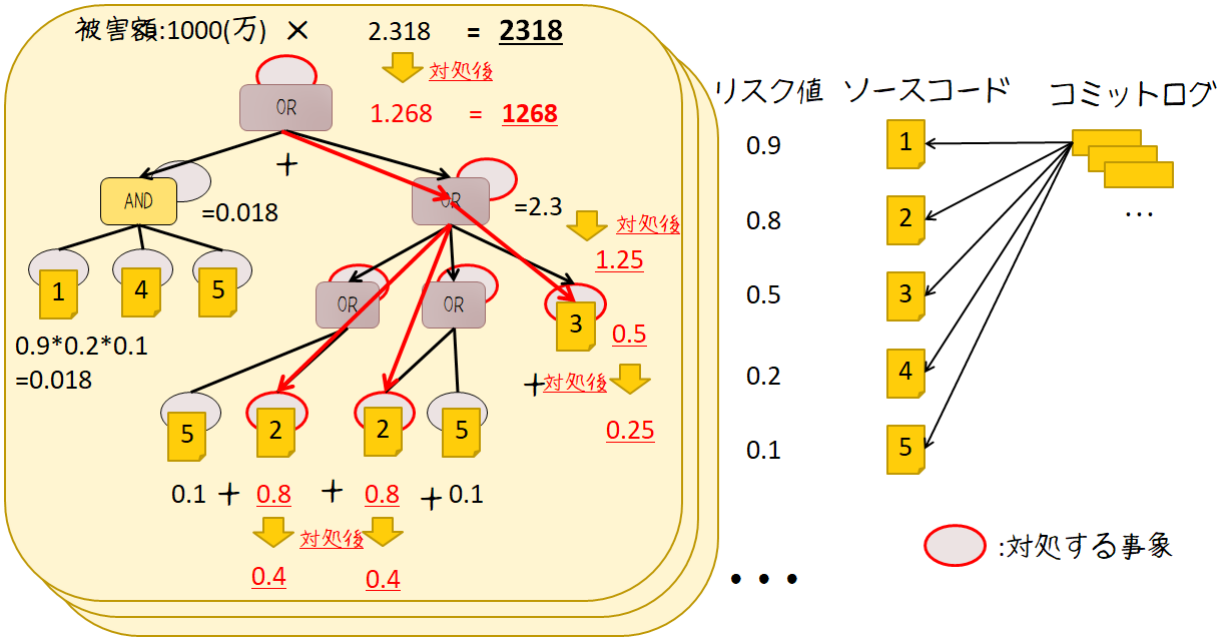


図 14: 被害金額期待値とリスク値を掛け算する

どのようなテストを行うかや、テスト以外のソースコードレビューなどで対応するかを検討する。リスク値によってどの程度のテストをするかについて、テストの重みづけとして規定しそれを参考にする(成果物 2_12_フォールトテストの重み表)。

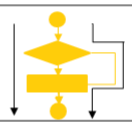
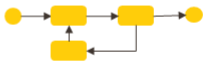
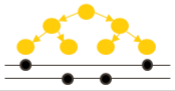
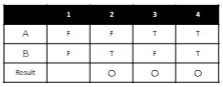
テストでどの程度リスクを下げられるかは、リスク値低限度を規定して計算する(成果物 2_12_フォールトテストの重み表)。リスク値低限度は、過去の不具合やテストの経験から設定する。最初は精度が低いことが想定されるので、テスト結果などからチューニングを進める

6. テスト詳細設計

6.1 フォールトツリーの原因ごとに、テストタイプを決定する

フォールトツリー図で根本的な原因を出しており、その原因を検出できるテストタイプ（技法）を選定する。

「テストタイプ-テスト技法対応表」として整理し、これを参考としながら、どのアプローチが最も有効かを検討する。対応表で機械的に マッピングはできないので、プロダクトのソースコードや仕様をみつつ、原因が生じる方法と、それをどの技法で検出すべきかの 2 段階に分けて検討を行う。

テストタイプ	(キーワード)	テスト技法	(テストケース導出モデル)
• データ (領域、境界、サイズ) • 性能	テストタイプ+ 抜け/漏れ ズレ/跳び 過剰/不足 矛盾/衝突 未反映/初期値/なし	同値分割 境界値分析	例 
• 流れ (シナリオ、処理)	同上	フローチャート	例 
• イベント/タイミング (状態、モード、遷移)	同上	状態遷移図/表	例 
• 構成 (環境、機器、モジュール)	同上	分類ツリー法	例 
• 条件 (入出力、制約)	同上	デシジョンテーブル	例 

	1	2	3	4
A	F	F	T	T
B	F	T	F	T
Result		○	○	○

図 15: テストタイプ・テスト技法対応表

6.2 テストケースの作成

テストタイプ（技法）を決めると、具体的なテストケースを作成する。

リスク値をもとにテストするかどうかを決めたフォールトツリー図（てすにゃん V3_成果物 2_08_フォールトツリー図_R04）をみて、決めたテストタイプに応じたテストケースの導出モデルを書く。

具体例として、フォールトのトップ事象の「課金ができない」-「演奏料金が設定できない」-「設定しても演奏料金額が特定の料金に反映されてしまう」を考える。

設定した演奏料金額がフォールトを起こす要因だと考え、テストタイプは「データ」である。リスク低減値のレベルは、落としたいレベルから「重」になる。

テストケースは、成果物 2_14_フォールトテストケースの「データ-１」シートに記述する。演奏料金額の他に課金形態と課金対象と予約登録数を組み合わせて、演奏料金が正しく設定されることをテストする。

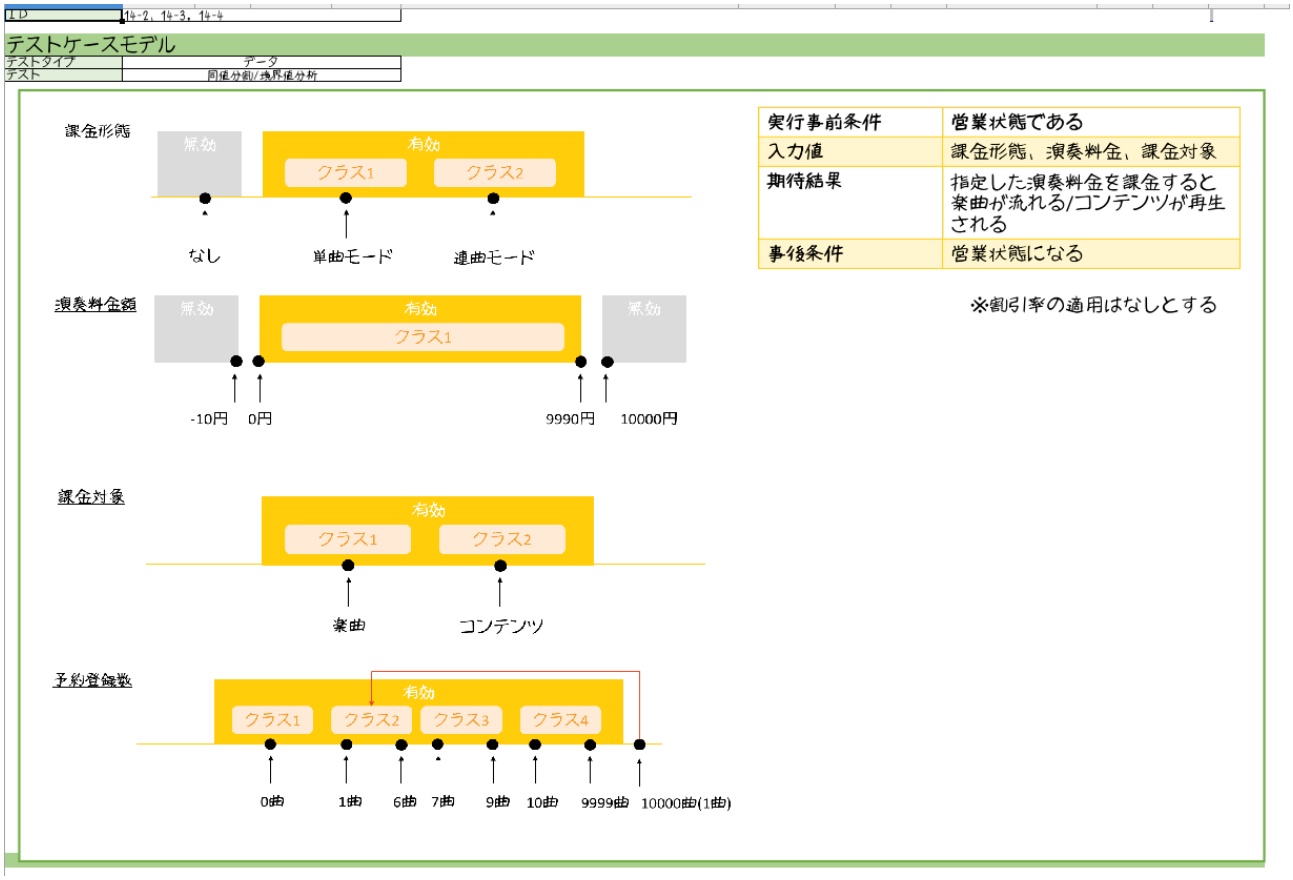


図 16: テストケース作成

7. 参考

7.1 用語集

用語	意味
フォールトツリー図(FT 図)	フォールトツリー解析(FTA)で利用される望ましくない事象に対して原因を表現する図のこと。てすにゃん V3 では FTA は使わないものの、似たような形式で記述するために同様に呼んでいる。厳密な FT 図の書き方には沿っていない。
フォールトテスト	てすにゃん V3 では、起こって欲しくないこと（リスク）のうち QA のテストでカバーするテストを指す。
リスク値	てすにゃん V3 では、Bugspots で出した値にリスクの対応状況を反映させた値のこと。
リリースクリティカルバグ	それを修正するまでリリースできない致命的なバグ。
ナレッジ	図「テストの全体像」でのナレッジは、開発者や QA の理解している製品要求やリスクに対する経験のこと。
フォールトベースドテスト	てすにゃん V3 では、フォートに対して QA で対応するテストのこと。
ユーザーテスト	てすにゃん V3 では、ユーザーが行うと決めたテストのこと。

7.2 参考文献

1. Eric S. Raymond 著、山形浩生訳(1999)『伽藍とバザール』
<http://cruel.org/freeware/cathedral.html>
2. Fogel Karl 著、高木正弘・高岡芳成 訳(2005-2013)『オープンソースソフトウェアの育て方』
<http://producingoss.com/ja/>
3. Jono Bacon 著、渋川よしき訳(2011)『アート・オブ・コミュニティ―「貢献したい気持ち」を繋げて成果を導くには』オライリー・ジャパン
4. Steve Freeman、Nat Pryce 著、和智右桂訳 (2012)『実践テスト駆動開発』翔泳社
5. ケント・ベック著、長瀬嘉秀監訳(2003)『テスト駆動開発入門』ピアソンエデュケーション
6. ジェームズ・A・ウィテカー、ジェーソン・アーボン、ジェフ・キャローロ著、長尾高弘訳 (2013)『テストから見えてくる グーグルのソフトウェア開発』日経 BP 社
7. 石原 一宏 、田中 英和、田中 真史著(2012)『【この 1 冊でよくわかる】ソフトウェアテストの教科書―品質を決定づけるテスト工程の基本と実践』SB クリエイティブ
8. 河野哲也『ソフトウェア要求仕様における HAZOP を応用したリスク項目設計法』ソフトウェアテストシンポジウム 2012
http://jasst.jp/symposium/jasst12tokyo/pdf/D2-1_paper.pdf

文末脚注

i あくまでコミュニティにおいてコードコミットなどの貢献しているフルタイムメンバーの数である。多くのプロジェクトでは、ほとんどは開発者であるため、LibreKaraoke においても大半を開発者との想定にしている。企業内のみで活動している人員は、コミュニティにとって直接価値を提供しておらず、また外から見えないために無視する。

ii オープンソースプロジェクトごとに、エコシステムや開発プロセスなどのコンテキストは大きく異なる。今回は筆者がもっとも馴染みのある LibreOffice のコンテキストに近づけている。LibreOffice について知りたい場合は Annual Report が最も簡潔にまとめられている。
https://blog.documentfoundation.org/blog/2017/07/21/annual-report-2016/

iii Debian Project では最近は安定版が約 2 年に 1 回リリースされており、次のバージョンがリリースされてから 1 年間メンテナンスするポリシーのため。ただしパッケージによってはメンテナが活動できていない場合も少なくない。

iv フォールトツリーの先端の原因を導出する作業は、プロダクトの作りとそれから発生する問題を検討することになる。テスト要求分析というよりも、テスト設計プロセスと呼ぶべきかもしれない。