研究コース 4（自動テスト）

フレイキーテストにどう立ち向かうか

How to deal with the flaky test

研 究 員：今村　大輔（株式会社オージス総研）
　　山田　高史（ブラザー工業株式会社）
主　査　：永田　敦（サイボウズ株式会社）
副 主 査：荻野　恒太郎（株式会社カカクコム）
アドバイザー：山口　鉄平（株式会社 LayerX）

研究概要
本研究はフレイキーテストの現象を理解し，効果的な対策とする実験を通じて，自動テストの高度化とプロダクトの品質向上を目指すものである．研究員の現場ではアジャイル開発に伴うテスト自動化を推進しており，一定の成果を挙げている．しかし，自動テストのテストケース数が増えるにつれ，無視できない頻度でフレイキーテストによるテスト失敗が発生し，ソフトウェア開発業務の運営に大小様々な支障をきたしている．
研究員のドメイン領域は制御系と WEB 系である．ドメインは相違しながらも，共通するフレイキーテストへの傾向と対策があることが明らかになった．そこで，制御系と WEB 系それぞれで発生している問題の詳細と対策実験を通じ，フレイキーテストに対するドメイン領域を問わない汎用性のある考え方を示す．

Abstract Through understanding the phenomenon of flaky test and experimenting with effective countermeasures, we try to advance automated testing and improve product quality.

1　フレイキーテストの問題点

フレイキーテストとは，テストの実行結果が実行するたびに異なるテストである．その原因として,テスト対象の状態やテスト環境の状態,テストの実行順序などが挙げられる．不具合の誤検出/見落としにつながり，テストの信頼性を低下させてしまう．結果，ソフトウェア開発業務の随所に影響を及ぼすと言われている．

フレイキーテストの研究では Google のテストエンジニアによる研究が有名である．Google においては全てのテスト実行の 1.5%においてフレイキーネスが発生しているとし，テストケースの 16%がフレイキーであると報告している．［1］

また，Spotify はフレイキーネスを可視化することで問題の解消につなげたことを報告している．縦軸に CI でのテスト実行結果を，横軸にテストケースをプロットすることで，視覚的にフレイキーネスを把握することができるとしている．［2］

2　フレイキーテスト対策方針を判断するフレームワークの提案

では，定量的な計測をもとに，フレイキーテストと判定されたテストケースに対して，対応方針はどうすればよいか.Google ではテストケースを隔離して実行対象から外し,「フレイキーネスを減らすための不具合」として管理している［1］．フレイキーテストかそうでないかの「判定」を重視し，「修正」に重きを置いていない．

残念ながら研究員の現場は Google や Spotify のような潤沢な資金・人員・設備を所持していない．フレイキーテストを判定するために膨大な回数テスト実行を繰り返し，高精度な統計を取るのも苦しいのが実情である．本論の読者層もおよそ同様であろう．

そこで本論では，リソースが慢性的に不足している現場向けに，簡易な統計から当たり

を付けたフレイキーテストに対して，シンプルに「原因」と「影響」のマトリクスで対応方針判断を行うことを提案する．

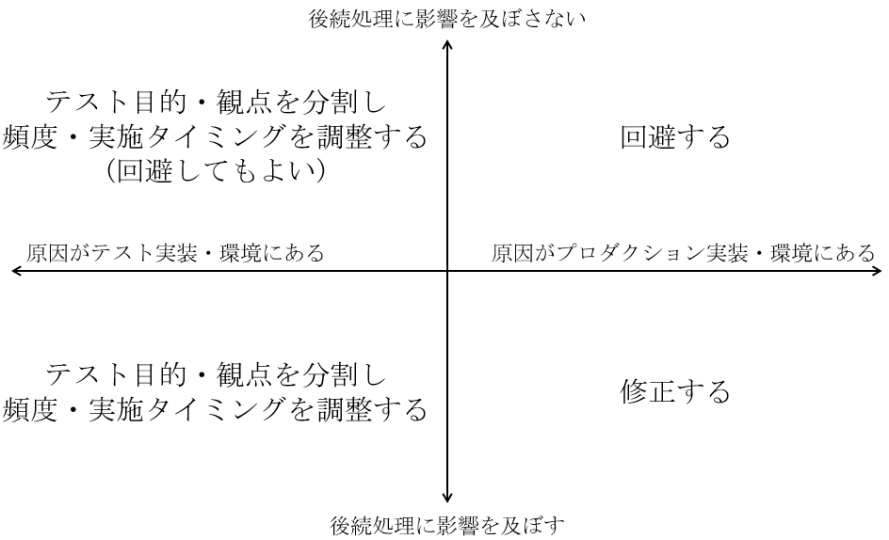

図 1 「原因」と「影響」のマトリクス

横軸でフレイキーネスの原因が「テスト実装・環境」か「プロダクション実装・環境」にあるかを分類し，縦軸でフレイキーネスが後続処理に「影響する」「影響しない」を分類する．その組み合わせで「限りあるリソースを効率的に活用できる対応方針」を示している．なお，ここで言う「後続処理に影響を及ぼす」は，およそ以下のようなケースをイメージするとよい．

- テスト対象機能の出力を別機能や連携先システムが入力として扱うケース
- テスト実行後の開発工程やアクティビティに影響を及ぼすケース

各象限のポイントは，以下の通りである．

1. テスト目的・観点を分割し頻度・実施タイミングを調整する
   - ✓ テスト側の問題であるため，フレイキーネスへの対応で直接便益を得るのは開発者(テスト担当者含む)である．積極的に対応する必要性は薄い．後続処理への影響を抑え込むことが重要である．フレイキーネスが発生する部分に対し，日常的に行うデベロッパーテスト(開発を促進する目的)と QA テスト(品質保証目的)を切り分ける．デベロッパーテストとしては高頻度実施し，フレイキーテストは回避する．QA テストとしては低頻度実施ながら，フレイキーテストも実施する．
2. テスト目的・観点を分割し頻度・実施タイミングを調整する(回避してもよい)
   - ✓ 1 とほぼ同等である．テストの問題，且つ，後続処理への影響がないことから，割り切って回避とするのもよいだろう．
3. 回避する
   - ✓ この象限はテスト対象の不具合であるが，フレイキーネスの影響がテスト対象に閉じている．後続処理で予期しないトラブルに拡大するリスクが無い．フレイキーネスの頻度にもよるが，優先順位は低い．単純なリトライ処理などで問題を回避し，優先すべき課題へ開発リソースを振り向けるとよい．
4. 修正する
   - ✓ この象限はテスト対象の不具合であり，フレイキーネスの影響がテスト対象に閉

じていない．後続処理で予期せぬトラブルに拡大するリスクがある．開発の状況次第であるが,優先順位は低くはない.不具合の修正対応が必要なものであろう．

つまり,「フレイキーテスト対策」にあたり，優先順位付けを通じて「テスト戦略としてのリソース効率活用」につなげることを意図している．

このマトリクスの背後にはアジャイル開発(特に XP)で重視される「YAGNI [3]」がある．リソースが慢性的に不足している現場のプロダクトが競争優位性を高めるには，ユーザニーズの本質を捉えた箇所にリソースを集中し,結果を左右しない枝葉末節は(よい意味で)あきらめることが重要である．フレイキーテストに対しても必須ではない手当ては避けてシンプルに保ちたい．厳密性を重視するあまり複雑なテストコードを実装して，後続処理に影響しない箇所を過剰にテストすると保守性を損ない開発生産性が低下する.ひいては,プロダクトの品質向上に逆効果である．

さらに，このフレームワークを機能させるために「自己組織化 [4]」が重要であることを挙げておきたい．アジャイル開発(特に Scrum)で重要視される要素であり，チームが自らの判断で問題を解決することである．大規模プロジェクトによくあるトップダウンで定められた定量的判断基準のようなものとは限らない．フレイキーテストに対し，無知ゆえの場当たり的な判断を行うと技術的負債が発生する．特に,「回避」を選択したならば不具合を見逃すリスクが発生する．逆に,「対応する」ために完全・複雑なテスト実装にすると保守性を損なう．直接・間接を問わず品質に対する信頼を棄損してしまう．だが，チームが一貫性ある判断を行い，適切に「問題ない」ことの説明責任を果たせるならどうか．仮に「回避」を選択したとしてもアジリティよくプロダクトの競争優位性を高める前向きな選択である．

そのためには，チームメンバー個々人が対応方針を最適に判断できなければならない．フレイキーテストに対する知識をチームで共有し，判断がもたらした結果を分析し，議論することを通じて，チームの特性を踏まえた判断基準・考え方として洗練・進歩させるのが対策の本質である．

3　現場におけるフレイキーテストの実情と対策実験

研究概要の項で述べた通り，研究員のドメイン領域は制御系と WEB 系である．さらに,開発チームにおける各々のロールは QA と設計・実装である．およそ異なるドメインで正反対の立場である．しかし，共にフレイキーテストが原因でソフトウェア開発業務の運営支障をきたしている．

それぞれの現場で発生している問題に対し，このマトリクスを適用する対策実験を実施した．

3.1 ケース 1：制御系の現場

研究員(制御系)の場合，プリンタを PC から制御するデスクトップ GUI アプリケーションのスモークテストを実施している．テストドライバが人間に成り代わり，アプリケーションをステップ実行する．テストドライバはステップごとにアプリケーションの振る舞いを監視し，期待値と合致しているかを判定するのが基本形である．

先行研究の枠組みに合わせて発生状況を分析すると，以下のような状況であった．

- 総実行回数：387 回
- フレイキーテストが原因と思われるテスト失敗回数：20 回
- テスト失敗確率：5.2%
- フレイキーテストが疑われるテストケース数：275 ケース

- 総テストケース数：816 ケース
- フレイキーテストが疑われるテストケース数の割合：33.7%

フレイキーネスが原因と思われるテスト失敗割合が 5.2%であり，Google の報告の 3 倍以上である．また，フレイキーテストが疑われるテストケース数の割合は 33.7%であり，Google の 16%に対し 2 倍以上である．

本テストはその時々で利用できる空き PC・プリンタで行われる．フレイキーネス発生時，テスト環境に以下のような特徴がみられた．

- 無線 LAN でプリンタを制御している
- PC に搭載しているメモリが少ない

テスト対象アプリケーションとテスト対象プリンタが無線 LAN で接続されている場合に，通信速度・安定性に問題がありプリンタ検索に影響を与える可能性が考えられる．また，テスト対象アプリケーションはプリンタに送信する印刷データを生成している．テスト実行されている環境で CPU やメモリの使用率が高くなると，印刷データ生成が遅延し，それに引っ張られてプリンタへのデータ送信が遅延する可能性が考えられる．

そして，この現場におけるフレイキーテストが引き起こしている問題は，テストのスケジュールが乱れることである．この自動テストはリリース前の最終検査として実行されており，リリース予定に対しクリティカルパスである．1 回の実行に約 40 時間かかり，フレイキーネスで fail した場合は最初から再実行しなければならない．締め切り直前でのフレイキーネス発生は，後続処理であるリリース判定，ひいては，市場投入スケジュールを乱してしまう事業部レベルの大きな問題である．

このような状況に前述のマトリクスを当てはめると，「原因がテスト環境にある」「後続の処理に影響を及ぼす」ことから「テスト目的・観点を分割し頻度・実施タイミングを調整する」となる．

つまり，具体的な解決すべき課題は

- リリーススケジュールを乱さない

となり，対策方針は

- スモークテストとして「基本的な機能が正常に動作すること」を確実に保証する
- ネットワークが不安定・メモリ不足等，悪条件のテスト環境での実施は別スケジュールに切り出し，クリティカルパス外で不具合検出に注力する

と設定できる．

この対策方針が機能するかを確認するため，以下の検証を行った．

1. 安定したテスト環境でスモークテストを実行することで，フレイキーネスが発生せず，予測した実行時間でテストが完了すること
2. 悪条件のテスト環境で実行することで，フレイキーネスを意図的に発生させ，詳細な調査ができること

検証の結果，フレイキーテストが作業進捗に影響を与えにくくなった．また，フレイキーネスの背後にある問題の調査がやりやすくなった．具体的な結果は以下の通りである．

1. 安定したテスト環境でのスモークテスト
   - 有線 LAN・メモリ増強した環境で実行することで，想定の 40 時間程度でテストを完遂できている
   - テスト失敗確率が 5.2%だったところ，1.4%まで下がっている．フレイキーネスが根絶されたわけではないが，リリース予定を乱すリスクを軽減できている
2. 悪条件のテスト環境での実行
   - 無線 LAN，且つ，メモリ資源を少なく限定した環境であれば，フレイキーネスが 7.7%の高頻度で発生する．即ち，問題を起こしやすくなった
   - 原因調査のための付帯スクリプトや動画記録をテストドライバと組み合わせて実行できるようになった
   - 無線 LAN でプリンタ制御とは別の大きなトラフィックが発生している場合に，プリンタが検索できない条件を絞り込めた

3.2 ケース 2：WEB 系の現場

研究員(WEB 系)の場合，Web API に対する E2E の自動テストを実施している．テストドライバがテスト対象へ HTTP 要求を送信し，テスト対象が HTTP 応答を返す．テストドライバは HTTP 応答の内容や出力されたログが期待値と合致しているかを判定するのが基本形である．

先行研究の枠組みに合わせて発生状況を分析すると，以下のような状況であった．

- 総実行回数：77 回
- フレイキーテストが原因と思われるビルドの失敗：42 回
- ビルドの失敗確率：54.5%
- フレイキーテストが疑われるテストケース数：46 ケース
- 総テストケース数：約 10,000 ケース
- フレイキーテストが疑われるテストケース数の割合：0.46%

フレイキーネスが原因と思われるテスト失敗割合が 54.5%であり,Google の報告の 36 倍である．また，フレイキーテストが疑われるテストケース数の割合は 0.46%であり，Google の 16%に対し著しく低い．

また，Spotify が提唱するように，フレイキーネスによるテストケースの fail を可視化することで，以下の特徴が読み取れた．

- ほとんどのテストケースで総実行回数のうち 1 度だけ fail が発生している
- 例外で，ある 1 つのテストケースのみ高頻度で fail が発生している

これらから読み取れることとして,(例外の 1 ケースを除き）個々のテストケースに特有の課題が潜在しているのではなく，自動テストの方式設計に課題があると言える．フレイキーネスが発生しているテストケースについて，具体的に fail している箇所を確認すると以下の共通点があった．

- テストドライバがテスト対象の出力するログファイルを監視して，期待したログが出力されているかを判定する

テスト対象アプリケーションのログ出力処理は OSS ライブラリが行っており，非同期でファイルに追記している．テストドライバがログファイルを監視して assertion をかけるタイミングと，テスト対象アプリケーションがログ出力を行うタイミングが同期している

保証はない．フレイキーテストの原因として大いに考えられる．

そして，この現場においてフレイキーテストが引き起こしている問題は，テスト実行毎に発生するクラウドサービスの利用料である．本テストの実行環境は AWS 上に構築されており，テストを全ケース実行すると$0.3 程度ほど課金される．また，Pull Request を承認する条件としてテスト全ケース pass を義務づけている．フレイキーネスで fail するとテストを再実行して Pull Request を依頼する必要があり，その都度課金が発生してしまう．テストの再実行は Pull Request ごとにおよそ 0 回～3 回程度発生している．さらに，この現場では月ごとのクラウドサービス利用料上限が定められている．万が一，上限超過が見込まれる場合，テスト実行回数を削減（場合によってはテスト実行を禁止）しなければならない.テスト自動化の目的は高頻度の回帰テスト実行を通じて開発を促進することだが，実行回数削減や禁止してしまうと目的を果たせない．

このような状況に前述のマトリクスを当てはめると，「原因がテスト環境にある」「後続処理に影響を及ぼす」ことから「テスト目的・観点を分割し頻度・実施タイミングを調整する」となる．

つまり，具体的な解決すべき課題は

- フレイキーネスによるテスト再実行に伴う無駄なキャッシュアウトを抑制する

となり，対策方針として以下のようにテストの目的・観点を分割して実行頻度を調整したものが導出できる．

表 1 WEB 系の現場で立案したフレイキーテスト対策方針 目的・観点分割・実行頻度調整

目的	テスト	観点	実行頻度
● 回帰結合テストとしてデグレード検知を早期・高頻度に行い，開発を促進	Web API E2E テスト	● 各 Web API が HTTP 要求内容に対し，期待する HTTP 応答を返すこと ● 各 Web API の処理において，データストアに期待する内容が登録・更新・削除されること	● Pull Request を起票する都度 ● 日次 ● (必要に応じて随時)
● システムテストとして運用・保守性を担保	ログ出力テスト	● 各 Web API が HTTP 要求内容に対し，期待するログ出力を行うこと ● 各 Web API が出力したログが欠損していないこと	● リリース前 ● (必要に応じて随時)

導出の考え方について補足すると，ログ出力自体は必須であるが，ユーザが便益を期待して利用する機能仕様そのものではない．副次的な非機能仕様である．Web API の機能仕様を検査するテストとしては必須ではないだろう．適切なログ監視ができるか，欠損なくログ出力されて監査証跡に使用できるかは運用・保守性に対するテストとして検査されるべきである．

したがって，日常的にデベロッパーテスティングとしてデグレードしていないかを検査する Web API E2E テストにおいて,フレイキーネスの原因であるログ出力内容の検査は「回避」してもよいと考える．フレイキーネスによるテスト再実行がなくなれば，クラウドサービス利用料のキャッシュアウトを抑制できる．

とは言え，単純に「回避」するのみでは不具合を見過ごしてしまうリスクが発生する．

そこで，ログ出力テストをシステムテストとしてリリース前に実行してヘッジする．

なお，Web API E2E テストとログ出力テストの分割は元々あったテストコードの assertion 調整のみ実現できる．ログ出力内容に対する assertion を「ログ出力テストの場合」のみ評価されるように修正すればよい．軽微かつ安全な修正の範疇だろう．

この対策方針が機能するかを確認するため，以下の検証を行った．

1. Web API E2E テストでフレイキーネスによるテストの fail が発生しないこと
2. ログ出力テストで処理内容に対し適切なログが欠損なく出力できたことを保証できること

検証の結果，Web API E2E テストではフレイキーネスを根絶でき，再実行に伴うキャッシュアウトをカットできた.また,ログ出力テストでは具体的な結果は以下の通りである．

1. Web API E2E テスト
   - フレイキーネスによるテストの fail が発生しなくなった.1 回の実行で全件 pass するようになった
   - Pull Request の際にテストを再実行する必要がなくなった．1 回の Pull Request にかかるテスト実施のクラウドサービス利用料が最大$0.9 程度から$0.3 程度に削減できた
2. ログ出力テスト
   - 何度かの再実行が必要であるが，フレイキーネスが発生しなければログ出力内容の検証ができている
   - 実行毎（フレイキーネスが発生してもよい）に出力されたログを比較して出力内容が合致していることから，非同期で出力されたログが欠損せず一貫性をもっていることを検証できている

過去実績で Pull Request は 1 年で 500 件ほど起票されているため，概算値で年間$300 のキャッシュアウト抑制となる．

また,「キャッシュアウトを抑制する」という課題設定には直接対応づかないが,Web API E2E テストでフレイキーネスを根絶したことにより，以下の大きな副次効果が得られている．

- フレイキーネスによる fail か意図しないデグレードかを調査する手間が省けている
- フレイキーネスという本質的ではない事由によるテスト再実行とその完了を待つストレスから解放されている
- 開発中バージョンの主ブランチや本番稼働中バージョンのリリースブランチに対する日次回帰テストでデグレードを見逃すリスクを排除できている

さらに，ログ出力テストにおいて，フレイキーネス発生有無に関わらずログ出力内容は同内容（トークン値など，実行毎に変化するデータ項目は除く）であることを確認した．フレイキーネスの原因はログ出力が非同期であるにもかかわらず，テストドライバが同期しているものとして出力内容を検証していたことと確定できた．

3.3 制御系と WEB 系の実験結果考察

制御系と WEB 系それぞれの実情と実験結果を考察して共通的に言えることは，フレイキーネスそのものが問題というよりも，フレイキーテストはソフトウェア開発業務の円滑な進行を妨げるということである．

よって、フレイキーテスト対策をテスト戦略で解決する方向性、それを汎用的な考え方として表した「原因」と「影響」のマトリクスは有用であると言える．制御系・WEB 系ともに課題解決につながる対策方針を導出することができた．

4　今後の課題

提案したフレイキーテストの対応方針を判断するマトリクスについて，ドメインが異なる・立場が異なる 2 つの現場において，実験を通じて有用性を確認できた．

しかし,「2 つ」ではサンプリング数として心許ないのは否めない．普遍的な判断のフレームワークとして機能させるためには,まだまだ多くの現場での実験が必要だろう.特に,「原因がプロダクション実装・環境にある」象限に当てはまる場合，どのような対策方針導出ができるのか，その方針は有用であるかについて，評価が必要だろう．他にも様々なシチュエーションで揉まれることを通じて，より的確な表現・判断基準へとブラッシュアップし，広く共有された考え方へと発展させたい．

あわせて、将来的にはフレイキーネスの発見・原因分析についても方法論として表現したい．本論ではフレイキーテストかどうかは簡易な統計から当たりをつける，その原因は繰り返しテスト実行した結果を読み解いて仮説立案するものとした．前述のマトリクスが対応方針判断の問題を解消できたとしても，アジリティ向上のボトルネックが発見・原因分析に残っているとも言える．

5　まとめ

本論では，フレイキーテストが引き起こす様々な問題に対して，対策案を「原因」と「影響」のマトリクスをもとに「テスト観点分割」か「修正」を検討することを提案した．限りあるリソースを有効活用することで、テスト戦略の最適化につなげることを意図している．提案したマトリクスの有用性を制御系と WEB 系の 2 つの現場で実験し、いずれも課題解決につなげることができた．

制御系・WEB 系などドメインを問わず，フレイキーテストは不可避である．完全なテストを追求するよりも，テストの目的・観点を分割して実行頻度・実施タイミングを調整するほうがテスト戦略上有効だろう．

参考文献

[1] J. Micco, “Flaky Tests at Google and How We Mitigate Them,” 27 5 2016. [オンライン]. Available: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html.

[2] J. Palmer, “Test Flakiness - Methods for identifying and dealing with flaky tests,” 18 11 2019. [オンライン]. Available: https://engineering.atspotify.com/2019/11/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/.

[3] M. Fowler, “Yagni,” 26 5 2015. [オンライン]. Available: https://martinfowler.com/bliki/Yagni.html.

[4] J. S. Ken Schwaber, “The Scrum Guide,” 11 2017. [オンライン]. Available: https://scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf.