目次

実践リスクベーステスト-PRISMAメソッド-

著:Drs. Erik P.W.M. van Veenendaal CISA/訳:藤原史和

システムテストフェーズに入る前のアクティビティつまり開発や単体、結合テストは遅延することはよくあることです。

その結果、システムテストフェーズにしわ寄せがゆき、大きなプレッシャーの中で実施しなければなりません。

しかし、テストをすること自体をやめたり、遅らせたり、ましてや手抜きテストなどできません。

さてあなたは、この状況をどう乗り切りますか？

目次

※Practical Risk-Based Testing - Product RISk MAnagement: the PRISMAの原著者から翻訳と公開に関する契約を締結し翻訳しています。

1.イントロダクション

システムテストフェーズに入る前のアクティビティつまり開発や単体、結合テストは遅延することはよくあることです。

その結果、システムテストフェーズにしわ寄せがゆき、大きなプレッシャーの中で実施しなければなりません。

しかし、テストをすること自体をやめたり、遅らせたり、ましてや手抜きテストなどできません。

実プロジェクトでは限られたリソースの中で最良のテストを実施するためにテストすべき対象を優先順位付けします。

システムのどこが最も注意しなければ行けない部分ですか？

答えは一つではありません、どの部分をテストするかという優先順位付けはリスクベースでなければなりません。

テストにどれだけリソースをかけられるかとテスト後に欠陥を発見されるコストとの間には関係がります。

リスクによっては段階的な本番リリースする場合もあります。

段階的リリースでの一般的なテストアプローチは、リリースできそうな重要な機能をテストし、他の機能は後回しにすることです。

システムレベルのテストでは、まず第一に製品やプロジェクトにおいて最も重要なことをテストする必要があります。

最も重要な場所を特定するには、機能の表示領域であるかや使用頻度、および故障した場合の想定されるコストを見て判断できます。

第二に、多くの欠陥を見つけられる場所をテストしなければいけません。

これは、製品内の欠陥のある領域を特定できれば判断できます。

プロジェクトの履歴を参照することでこれらを判断する何らかの手がかりを見つけられるでしょう。

製品の変更履歴からはそれ以上の手がかりを得られます。

プロジェクトの履歴、製品の変更履歴の両方を駆使して、テストを注力すべき領域とテストを省く領域のプライオリティリストを作ります。

テスト実行中にいくつかの欠陥が見つかった場合、欠陥を分析することでさらに集中すべきところや、テストの方法を調整できます。

その考え方は、欠陥が発生しやすい領域では欠陥が偏在しており、その欠陥は開発者が作り出した特定の問題が症状となって顕在化するからです。

そのため、欠陥は近くに欠陥が多く、おなじような欠陥が多いという結果に至ります。

したがって、テスト実行フェーズの後半部分では、欠陥が発見された領域に焦点を当て、前に検出された欠陥と同様の種類を検出するためのテストを実施する必要があります。

2.製品リスク管理

本稿では、最もテストしないといけない重要な領域やリスクが最も高い項目を見つける方法について説明します。

プロダクトリスクマネジメント（PRISMA）メソッドが開発されたあとこれはさまざまな産業で多くのプロジェクトや企業において使われています。

この方法はリスクベースのテストを行う際特に、リスクの特定や分析においてステークホルダーに協力を得るためにテストマネージャをサポートします。

PRISMAメソッドは、あらゆるテストレベルで使用できます。

それにはコンポーネントテストや統合テスト、システムテストまたは受け入れテストが含まれます。

組織レベルとプロジェクトレベルの両方で使えます。

組織レベル全体に適応した場合、このメソッドを使用して、組織内のほとんどのプロジェクトまたは開発プログラムに共通する資産に対して扱えます。

適応した結果は、プロジェクトによって適用される青写真として使え、また全体的なテスト戦略の一部としてドキュメント化できます。

プロジェクトレベルに適応した場合、テストプランとして記述されているプロジェクトのテストアプローチのプロダクトリスク分析として使えます。

注意しなければいけないこととしてはPRISMAはプロジェクトリスクではなく製品リスク管理のための方法であることに気をつけてください。

実プロジェクトで詳細なテストアプローチを決定するには、製品リスクとプロジェクトリスクを組み合わせて決定する必要があるでしょう。

製品リスク分析は、適切なテストアプローチを決定し、最もリスクが高い項目が、リスクの低い分野よりもテストフェーズ初期に集中的にテストされるようにテストを設計するために使用します。

製品リスク分析の結果によっては開発手法にも影響を与える可能性があります。

例えばテストフェーズの初期段階でのテストを行うように優先づけしたり、経験が豊富なエンジニアをより高いリスクの領域の開発にアサインするといったことができるようになります。

リスクの追跡は、テスト設計フェーズだけではなくプロジェクト全体を通じて行う必要があります。

なぜならばリスク分析を定期的に繰り返すことで、初期リスク分析が正しかったか検証できます。

また、報告を含む検査監視と制御は初期リスクで発見された部分周辺で組織的に行わなければいけません。

多くの場合初期にリスクを特定して、プロジェクト中では一度も監視されていないということがしばしば見うけられます。

> プロダクトリスクマネジメント（PRISMA）メソッドが開発されたあとこれはさまざまな産業で多くのプロジェクトや企業において使われています。この方法は、リスクベースのテストを行う際にテストマネージャをサポートします。

> 「十分に良い」というのは、テスト理論への形式主義に対する答えです。
>
> "欠陥0"を目標とすることはソフトウェアにおいては合理的ではありません。
>
> あなたは自分自身に対してはもちろん、ユーザや顧客に対して"欠陥0"を目指していると偽ってますか？そうでないでしょう。

3.十分なテスト

リスクベースのテストやこの本で説明したPRISMAメソッドは、ジェームズ・バッハは1997年に考えた「十分なテスト」（Bach、1997）というコンセプトに非常によく関連しています。この考えは当時テスト・コミュニティを二分させました。

テストの専門家の片方の言い分としてはそれは、責任回避だとか妥協だとか単純すぎて有用ではないと主張し、さらにこれによって面倒な作業が発生するとも言っていました。

その一方リスクベーステストの支持者は、そもそも「完璧な」解決方法はなく、我々が実プロジェクトの中でできる現実的な方法だと主張しました。

十分なテストのアプローチは、リスクベースのテスト手法の理解を助けます。リスクテークされているプロジェクトにおける「リリースジャッジ」のための良いフレームワークです。

あなたがテスターとして次のように尋ねられたことはありませんか？「システムをリリースに十分なテストを行ったのですか？」と、このような大きな決断を下すときに、どうすればその質問に答えられますか？

プロジェクトマネージャーから言わせれば「テスターは"う〜ん、まだ十分ではないですね"といつも言っているのでテスター達はいつまでたっても満足することはない」と思っています。プロジェクトマネージャーはテスターを悲観主義者として認識します。

もしあなたが「まあ、大丈夫でしょう」と言うとプロジェクトマネージャーはあなたの目の前に紙を突き出してサインオフ（署名)をしてくださいと頼むでしょう。

そのサインオフ(署名)をしたら、あなたは他人の責任を負えますか？

つまり「十分に良い」とはどのような状態でしょうか？また、リスクベースのテストはこの問にどのように答えてくれるでしょう？

「十分に良い」というのは、テスト理論への形式主義に対する答えです。

"欠陥0"を目標とすることはソフトウェアにおいては合理的ではありません。

あなたは自分自身に対してはもちろん、ユーザや顧客に対して"欠陥0"を目指していると偽ってますか？そうでないでしょう。

あなたの顧客やユーザーは現実世界に住んでいるのです。妥協は避けられません。あなたはいつも次から次へと現実が押し寄せ。そしてその先には不完全な情報に基づいて決定を下すことに挑戦しなければなりません。

テスター一として見積もりを減らしたり、テストの実行対象を絞り込んだりしても、どうか怒らないでください。

なぜならばどんなプロジェクトにおいても予算とリソ一スは常に制約されており、欠陥をゼロにすることはできないため、テストを絞り込む罪悪感と恐怖にさいなまれることは避けることは出来ないはずです。

システムをリリ一スする（または増強させる）という文脈において「十分」の定義は次のとおりです。

1.現在の状態で十分に価値を出せる事。

2.重大な問題がない事。

3.その機能が作る価値は、残存する問題（重大ではない）が引き起こす可能性のある損害よりも十分価値がある事。

4.現在の状況、考えられているすべての事柄において、それを修正するためにリリ一スを遅らせたならば、利益よりも損害の方が大きい事。

この定義は、このシステムが本番で稼働し、価値を得、利益を得るために、十分に働いていること（このシステムの拡張を含む）を意味します。

「重大な問題はない」とは、使用不能または容認できなくなるような重大な障害を引き起こすような欠陥はないということ意味します。

現時点ではすべてのことを考慮して、完璧を目指して時間やお金を注ぎ込むよりも、既知の問題を抱えたまま早期にリリ一スする方がコスト的にメリットがあります。

このフレ一ムワ一クは、利点が価値があると期待されている不完全な製品を時間通りにリリ一スすることを可能にします。

では、リスクベ一スのテストは、この「十分に良い」アイデアとどのように適合しているのでしょうか？

まず第一に、十分な利益が得られていますか？を確かめないといけません。

私たちが実行するテストでは、最低限利益を提供する機能が完璧に動いている事を実証する必要があります。

私たちはこれを証明しなければなりません。

第二として、重大な問題はありますか？という問いに答える必要があります。

インシデント報告つまりソフトウェアの欠陥の記録は、少なくとも重大な問題（および可能な限り多くの他の問題も含む）の証拠を提供しています。そしてこれが十分であるために重大な問題はないはずと言えるのです。

第三にテストは決定を下すのに十分か？という事です。

テストによってリスクに対して修正され、利益がリリースに対して利用できると言えるような十分な証拠を提供しましたか？

本質的にはこれらの全ての質問に対してバランスを取ることが必要です。

十分な品質と許容可能なレベルのリスク抽出する事を提供するという目的で、テストにリソースを費やすのです。

図1：テストと品質とリスクのバランス

誰がバランスを決定するか?

製品が十分かどうかを判断するのはテスターの責任ではありません。

ここで一つ例え話を入れましょう。テスターを裁判における証人と見立ててみましょう。

このシーンの主なプレイヤーは次のとおりです。

- 被告人（システム）
- 裁判官（プロジェクトマネージャー）

- 陪審員（ステークホルダー）
- 証人（テスター）

今回の例え話では弁護士の役割出て来ません。前提として裁判官と陪審員両名とも、証人からの証拠を抽出し、「事実」に対して議論する事に優れていると仮定します。

証人の役割に焦点を当ててみましょう。

証人は平等主義者（陪審員）に理解を促すために形式化され複雑な証拠を提示し説明するために法廷に連れて来られた人たちです。

裁判官は、客観的かつ役割として分離されたものでなければいけません。

証拠から有罪または無罪どちらが言えるかと尋ねられた場合、証人は証拠に基づいてどのように論理を展開できるかこれから説明しますが、判断そのものは裁判官に委ねられています。

この例えのようにテスターは、「これらの機能は動作し、これらの機能は動作せず、このリスクに対しては修正されており、このリスクは残っています」という証拠を表明するだけです。

これがシステムを受け入れられるかどうかはテスター以外の他の人が判断することです。

テスターは、ステークホルダーに決定を下すための情報を提供するために存在しています。

結局のところ、テスターはソフトウェアを開発したり欠陥を作り込むことはありません。

テスターはシステムを本番稼働できるかどうかを受け入れると言うリスクを冒しません。

テスターは、マネージャーや同僚に対して情報を提供するために独立した視点を提示する。

製品がリリースに対して十分かどうかを判断するように尋ねられたとき、テスターは得られた証拠に基づいてこれらの利点が利用可能であると発言する可能性はありますが、リスクは依然として存在するのです。

しかし、実プロジェクトにおいてテスターとして決定を下すように求められたら、どうしたらいいですか？

答えは、ステークホルダーが決定を下すのを最大限手助けしなければならないということです。

例えば6ヵ月前にテスターから報告された、機能不全に陥れるような問題は現時点でまだ存在するかもしれません。

当時のステークホルダーとテスターの合意が得られているとするならば、リスクの認識を緩和しない限り、システムをリリースことはできません。

未解決のリスクについて以下の情報から判断が下される必要があります。

- 特定のリスクに対して修正されたと判断するのに十分な証拠がある事。
- 一部の機能が機能しない（リスクが発生している）という証拠がある事。
- 証拠の欠如（テストが実施されていないか、テストが計画されていないため）のリスク（の疑い）が残っているという事。

これは判断として理想的ではないように思えるかもしれませんが、

先述した非現実的な理想的な基準よりもましだと言えます。

あなたはそれでもなおシステムのリリース可否について意見を述べることを余儀なくされるかもしれませんが、私たちは、この原則的な立場をつまり裁判における証人という立場を、プロジェクトの早い段階できっちり表明することよって、マネージャーへの信頼性を高めると信じています。

マネージャーは、将来のプロジェクトで適切な責任を負う可能性があります。

テストフェーズにおけるよくあるプレッシャー

これは開発者がテスト開始予定日よりも遅れてテスターに引き渡される時に発生します。

リリース日は固定されたままで変わりません。

開発フェーズ終了の基準は、テストフェーズが開始できるかを判断するために使われますが、開発フェーズが制限時間内に完了されない場合も多々あるかと思います。

しかしながら、リリースへの圧力は非常に大きく、この基準を脇に置きがちです。

この場合重大な欠陥があったりダウングレードしたりする可能性があります。

フェーズ終了判定は、プロジェクトの初期段階において理想的に考えられ制定されますが、これは残念ながら意思決定を容易にするものではありません。

テストの開始時になって明らかになったリスクが、全体にわたって見えています。

テストフェーズが短縮された場合でもリスクベースのテストをしていれば、いくつかのリスクに対して修正出来た可能性があり、テスターが新たなリスクを明らかにした可能性はもちろんありますが、少なくとも危険性の高いリスクはすべての関係者にとって明らかになっています。

実際にリリースする決定が下された場合、ステークホルダーは、すべてのリスクに対して修正し、すべての利点が利用可能であるという証拠がだされる前に、製品がリリースできるということを明示的に選択しました。

リスクベーステストは単にテスターに対してだけに利益をもたらしません。ステークホルダーに対しても、その決定を下すのに十分な情報があると判断出来ました。

実際、リリースの決定を行うための情報は、テスト実行フェーズの初日から利用可能になります。

テストの証拠と残存リスクのバランスはリリース判定を大きく左右します。

したがって、リスクベースのテスターにとって肯定的な（しかし驚くかもしれないが）この原則は、テストフェーズの期間は、あなたの「良いテスト」を行う能力には関係していないということが言えます。

4.テストフェーズにおける課題

もし、あなたが一人のテストマネージャーとしてアサインされている場合どんなシナリオが考えられるでしょうか？

あなたはこのテストのための計画と予算を策定するミッションを負っています。

最初は妥当であり十分に考えられた計画を策定していました。

さてテストフェーズが近づくと、開発チームから製品がまだ準備が十分ではなく、テスト開始に間に合わないと報告をうけました。

時間的な制約であなたのテスターはいくつかのテスト実施ができないことがわかりました。

もちろんテスト期間短縮に反対したり、間に合うように言ったりもっとテストにリソースを割り当ててほしいと主張することはできますが、この主張が通るとは限りません。

つまり想定よりも少ない予算と時間でできるテストをしなければなりません。

あなたは可能な限りこの製品に対してテストしなければならず、本番で問題なく動作することを確認する必要があります。

この状況をどう乗り切りますか？

もしあなたがテストを手抜きしたならば、それは品質が規準に達していないことに起因する問題が発生するでしょう。

一方もし、予定していたテストをすべて消化するように行動したなら本番リリースはあなたのせいで遅れるでしょう。

ここでは建設的な解決策が必要になってきます。つまりゲームを変更する必要があるのです。

マネージャーが理解できるような説明の仕方で、あなたが持っているタスクのうちどのタスクが不可能かマネージャーに説明しましょう。

同時に代替案を提示することも重要です。

そしてマネージャーは製品をリリースするミッションを追っていますが、同時にリリースした場合のリスクを理解する必要もあります。

1つの戦略として、適切な品質レベルを見つけることでしょう。

製品においてすべての欠陥を修正する必要はありません。

同時に必ずしもすべての機能が完璧に動作する必要はないでしょう。

場合によっては、部分的に製品品質を低下させることもオプションとして考えないといけません。これによって、あなたは

重要性の低い部分のテストを削減できます。

もう1つの戦略としては優先順位付けするということです。まず最も重要な欠陥を見つけるためにテストを優先順位付けしましょう。

大体のケースにおいて最も重要なことと「最も重要な機能」はほぼおなじ意味です。

「最も重要な機能」はどの機能がシステムの主目的を担っているかを分析し、その結果どの機能が重要であり、どの機能が重要でないかを調べることで見つけられます。

これによって多くの欠陥が予想される場所をテストするなんてこともできます。

もしも製品の最悪の領域を見つけられたならば、その周辺に対して重点的にテストを行えば、より多くの欠陥を検出できるでしょう。

多くの重大な欠陥が見つかった場合、マネージャーはテストにもっと多くの時間とリソースを割くように考えるでしょう。

実際のプロジェクトでは、最も重要なもの（セクション5で議論されている）と最悪のもの（セクション6で議論されている）の組み合わせて考える必要があるでしょう。

リスクベースのテストでは、特にチームがテストを中断する必要がある場合は常に、利用可能な時間内で最良のテストを行っているという状態を作らなければいけないことに注意してください。

> システムレベルでは、まずアプリケーションで最も重要なことをテストする必要があります。
>
> これは、機能の可視性、使用頻度、および欠陥が市場流出した場合のコストを総合的に考慮して判断できます。

5.製品の最重要部分を見つける

テストとは常にサンプリングであり、すべての状態をテストすることはできません。つねにテストできる箇所は無限に見つけられます。

したがって、どのプロジェクトにおいてもテストするものとテストしないものに関して意思決定を行う必要があります。どこを重点的にテストするか、どの部分はそうでないのか。

テストフェーズの目標は、最悪の欠陥をテストフェーズの初期に見つけ、可能な限り多くの欠陥を見つけることです。

これを確実に行うためには、製品の中で最も重要な機能や特性を見つけだす事が非常に重要です。

できるだけ多くの欠陥を見つけることは、製品の欠陥が多く集まる領域に対して重点的にテストすることが重要です。

言い換えるならば、より多くの欠陥が埋め込まれていると予想される場所を知る必要があります。

これについては次のセクションで説明します。

まずは、製品の最も重要な領域を知る必要があります。このセクションでは、製品の領域を優先順位付けする方法について説明しましょう。

ここで紹介する方法は唯一無二のものではありません。製品によっては色々な要素があるかとおもいますが、ここに挙げられている要素は多くの製品やプロジェクトで利用できるでしょう。

重要な領域は、特定の機能や機能グループもしくはパフォーマンス、信頼性、セキュリティなどの品質特性として定義されるでしょう。

この本では、領域を一般的な用語「テスト項目」と呼んでみましょう。

テスト項目の重要性を判断する際に考えられる要素は次のとおりです。

クリティカルエリア（故障した場合のコストが大きいもの）

製品をとりまく様々な環境で製品を使用し、故障する可能性のある使い方を分析する必要があります。

そのようなクリティカルな故障を引き起こす可能性のある使いすべて、最低でも最悪の使い方を見つけます。

冗長性、バックアップ機能やまた、ユーザー、オペレーター、アナリストが使うであろう使い方を考慮しましょう。

レビューされていなくて、実際にプロセスに対して直接インストールされた製品は利用する前にちゃんと出力をレビューされている製品に比べてクリティカルでしょう。

製品がプロセスを管理する場合、このプロセス自体が分析対象となります。

ランク付けすると次のようになるでしょう。

- ランク1:故障によって製品が壊滅的な状況になってしまう事象:例えばこの故障によってシステムが停止されたり、環境内のタスクをストップさせる（ワークフロー、ビジネスや製品全体を停止させうる）可能性があるもの等です。また故障によって莫大な金銭的損失を生じさせたり、人命に対して損害を与える可能性のあるものも含まれます。
- ランク2:故障によって製品に損害を与える事象:製品全体が停止することはありませんが、データ欠損をおこしたり、破損したり、プログラムやコンピュータが再起動されるまで機能不全になる可能性があるものが含まれます。
- ランク3:故障によって動作が妨げられる事象:例えばユーザーの仕事を停止させたり、難しい手順を実行しないと同じ結果に到達できないもの等です。
- ランク4:故障によってユーザの迷惑になる事象:この故障は機能には影響しませんが、ユーザーや顧客にとっては望ましくないものが含まれます。

もちろん、製品毎に故障が与える影響は様々です。

たとえば（人間の）安全性に関連たり、財務上の損害を与えうる製品対しては故障は非常にクリティカルです。もう1つの方法としては、マーケティングの視点を使うことで重要性を判断できるでしょう。

たとえばこの製品の（ユニークな）セールスポイントは何かを考えることです。

表示領域

もしも何か故障が生じた場合、表示領域は、多くのユーザーに対して障害を与えうる領域です。

ユーザとして考えないといけないのはスクリーンの前に座っているオペレータだけではなく、出力されたレポートや請求書などを見ているエンドユーザ、またはソフトウェアを含むこの製品によって提供されるサービスに依存している製品のユーザの場合もあります。

この章では、そのような故障に対するユーザーの許容範囲を考慮すべき要因として考えないといけません。

様々な機能や品質特性の重要性に関するものに関しては、前の章を参照してください。

使用方法に関して訓練を受けていない、または使い慣れていないユーザー、特に一般の人々を対象としたソフトウェアは、UIに注意を払う必要があります。

頑健性(ロバストネス)も大きな懸念事項になります。

ハードウェアや生産プロセス、ネットワークなどと直接繋がって作用するソフトウェアは、ハードウェア障害やデータのノイズ、タイミング問題などの外部の影響を受け易いでしょう。

この種の製品は、環境が変化した場合を考慮して徹底した検証や有効性の証明、再テストが必要です。

可視性に関しては、外部の可視性（組織外）と内部可視性が区別されることが多く。それによって自分のユーザーだけがその問題を経験しているのかどうかということが分けて考えられます。

最もよく使われているエリア

製品の中で一部の機能は毎回使用されているのに対して、一部の機能は数回しか使用されていないということがあるでしょう。

また一部の機能は、たくさんのユーザが利用しているのに対して、一部の機能は少数のユーザしか使ってないということがあるでしょう。

使用される頻度に応じて機能に優先順位を付けます。

特定の機能領域のテストをスキップするためには優先順位付けが必要です。四半期に一回とか半年に一回または一年に一回だけしか使われない等例えば使用頻度で決められます。

このような使用頻度の少ない機能は、「最初にリリースしてしまってみんなが使用する前にテストする」ということも可能でしょう。

しかし、使用頻度分析が明確にできないこともあります。

そのような例としては、例えばプロセスを制御するようなシステムにおいては、特定の機能が外部から見えない場合があります。

そのような場合の対処方法としては、システムの設計部分まで詳細に見ることで分析できます。

こちらもランク付けすると次のようになるでしょう。

- ランク1:不可避な領域：通常利用シーン（スタートアップ、印刷、保存など）中にほとんどのユーザーが接触するであろう製品の領域。
- ランク2:頻繁に利用される領域：ほとんどのユーザーが最終的には接触する製品の領域ですが、特定の利用シーンに関しては接触しない可能性がある領域。
- ランク3:時々利用される領域：一般ユーザーに関しては一度も訪れたことない領域かもしれないが、より専門的、もしくは使い慣れたユーザーにとっては必要とする機能を扱う製品の領域。
- ランク4:ほとんど利用されない領域：ほとんどのユーザーが一度も訪れることのない領域もしくは、特定のユーザーが特定の行動を取る場合にのみ接触する、製品の領域。

これとは別で重大問題を引き起こす故障は依然としてクリティカルです。

重要な要件を選択するための別の方法としては、（A Cost-Value Approach for Prioritizing Requirements Karlsson et al 1997）を参考にしてみてください。

> テストの実行が開始された後、いくつかの欠陥が見つかった場合、これらを分析してテストにさらに集中し、テストの方法を調整できます。

6.製品の最も悪い部分を見つける

最悪の領域とは多くの欠陥が隠されている領域です。

つまり、多くの欠陥がどこにあるかを予測することが重要なタスクです。

これは、想定される欠陥発生要因を分析することで予測できます。

このセクションでは、最も重大な欠陥発生要因と欠陥が発生しやすい領域によく現れる症状について説明します。

欠陥は特定の要因に集中しています。これに関しては一部この章でも説明します。

それは最も重要な部分と最悪の領域を特定するために適用されます。

複雑な領域

複雑さというのはおそらく最も注意しなければいけない欠陥発要因です。

世の中には200種類以上の複雑さを計測する方法が存在し、複雑さと欠陥頻度(欠陥密度)の関連性についての研究は20年以上に渡って続けられています。

しかし、現在になってもなお一般的にちゃんと検証された欠陥の予測方法は存在していません。

それでもなお、現在使われている複雑さの指標のほとんどは実際に問題のある領域を明らかにできます。

指標の例としては、使われている変数の数、ロジックの複雑さ、制御構造の複雑さなどです。

複雑さに関してさまざまな側面に対していくつかの分析を行うことで、製品中で問題を引き起こす可能性のある領域を見つけられます。

変更領域

変更は重要な欠陥発要因です（Khoshgoftaarら、1998）。

その1つのケースとして、

変更は簡単であると主観的に認識されている箇所に関して、しばしば影響範囲が十分に分析されていない事があります。

もう一つは、時間の経過とともに徐々に変更が繰り返された場合、影響度分析が不完全なケースがあります。

その結果、副作用が起きてしまいます。

一般はソースコードの変更のログが存在するでしょう。

これは構成管理システムの一部と呼びます（存在する場合）。

変更を機能領域別にソートしたり、変更した行数が異常である領域を見つけられます。

これらの情報から、最初から設計が十分でない、または多くの変更によって元の設計が跡形もなく破壊され、さらにその後の設計が不十分である兆候が見て取れます。

変更した行数が異常であるというのは、設計が不十分である一つの兆候です。

つまり、大幅に変更された領域は、ユーザーが期待していないような欠陥が存在する可能性があります。

新技術や新しい方法を適用した領域

新しいツール、方法、技術にチャレンジしている場合、プログラマーは一定の学習曲線に沿ってスキルアップしていくでしょう。

学習曲線の初期では、それ以降と比較して多くの不具合を生み出すかもしれません。

新しいツールの例としてはCASE（Computer Aided Software Engineering）ツールや、会社内で新しく業務に取り入れるツール、市場においても新しくて不安定なツール等が含まれます。

もう一つの問題はプログラミング言語です。言語によってはプログラマーにとって新しいものもあるでしょう。

また、新しいツールやテクニックそのものが、問題を引き起こす場合もあります。

考慮すべきもう一つの要素は、利用する手法やモデルの成熟度です。

ここで言う成熟とは、理論的根拠の強さ、または経験的証拠の強さを意味します。

もし、対象のソフトウェアが、有限オートマトンや文法、関係データ・モデル等によって、適切に表現されている場合、それはかなり信頼できるものと期待できるでしょう。

一方、新しくてまだ実証されていない種類の方法またはモデル、または最先端技術が用いられている場合、ソフトウェアはより信頼性が低い可能性があります。

ほとんどのソフトウェア・コスト・モデルでは手法やツール、技術に対してプログラマーの経験に基づく要素が含まれています。

これは、開発コストの見積もりとではもちろん、テスト計画においても重要です。

多くの人が関与している領域

ここのアイデアは千匹の猿シンドロームと呼ばれます。

つまり、タスクに関わる人が多くなればなるほど、コミュニケーションのオーバヘッドは大きくなり、その領域が悪くなる（欠陥を多く含む)可能性も大きくなります。

少人数で熟練したメンバー達で作業するほうが、中程度の能力メンバーが集まった大規模なグループよりもはるかに生産的です。

COCOMO（Boehm、1981）ソフトウェアコストモデルでは、メンバーサイズはソフトウェアサイズの次に重要な要因となります。

それらの欠陥の影響の大きさは、欠陥を検出し修正するときにかかるコストから算出できます。

比較的多くの有資格者で構成されている領域は、よくテストされていると認識されることがあります。

この文脈において「有資格者」とは何かを定義することが重要です。

プログラミング言語のスキルやドメイン知識、開発プロセスのスキル、職務経験など。

その分析には注意が必要です：

いくつかの企業（Jørgensen、1984）では、より複雑な領域に対して最高の人材で対応し、はレベルの低い人材は簡単な領域にアサインしています。

しかしながら、欠陥密度は、人数または構成されるメンバーの優秀さに関連していないかもしれません。

優秀だけど欠陥密度が高い、典型的なケースとしては、大人数で、しかもフォローアップが十分にされていないコンサルタント達が開発しているプログラムです。

欠陥が多い理由の一つは彼らはそれぞれ全く違うやり方で働いているからかもしれません。

時間的プレッシャー

時間的プレッシャーにさらされると人はショートカットしたくなります。

人々は問題を解決することに集中し、しばしばすべてがうまくいくと楽観的に考え、品質管理活動をショートカットしようとします。

成熟した組織ではこの楽観主義がちゃんとコントロールされているでしょう。

また、時間的プレッシャーは、残業を誘発する可能性があります。

しかし、よく知られている事として、長期間の作業は人の集中力を奪います。

不十分なレビューとインスペクションは欠陥密度を極端に増加させる可能性があります。

開発中の時間的プレッシャーをデータとして取得したい場合、労働時間のリストの調査やマネージャーやプログラマーにインタビューすることによって取得できます。

図2：プレッシャー下で事態は悪化する。

楽観主義

COCOMOコストモデルでは、コスト要因の1つとして機械時間とメモリの不足があげれています。

論点としては、最適化をすると追加でデザインする労力を必要とするか、あまり堅牢でないデザインを使用して最適化を行えるということです。

追加デザインは、欠陥除去活動(バグフィックス)からリソースを奪う可能性があり、あまり堅牢でないデザインは、より多くの欠陥を生み出しかねねません。

欠陥の履歴

欠陥修復は"変更"をもたらします。これは先述したとおり、新たな欠陥を生み出しやすくなります。

したがって欠陥のある領域はこの負のサイクルを通して持続する傾向がある。

経験則によると、本番環境のシステムで欠陥のある領域は、レビューやユニットとサブシステムのテストで欠陥のある領域に遡れます。

次の2つの研究（Khoshgoftaar et al、1998）と（Levendel、1991）によると、過去に欠陥があったモジュールは将来に渡って欠陥を継続しやすいことを示している。

設計とコードレビュー、およびユニットとサブシステムのテストによる欠陥の統計が取得できたならば、後工程であるテストフェーズで優先順位を決定するのに有益なデータとなり得ます。

地理的に分散したチーム

プロジェクトで一緒に働いているメンバー同じ場所で働いておらず、地理的に分散ている事は、コミュニケーションの悪化をもたらします。

これは同じ建物レベルでも当てはまります。

地理的距離がプロジェクトに悪影響を及ぼしかねない場合、ある程度効果が判明しているアイデアは次に列挙するものです。

- 同じ建物の異なる階にオフィスを構えている人は、同じ階にいる人と同じくらいコミュニケーションをとりましょう。25メートル以上離れて座っている人は、十分にコミュニケーション取れていないかもしれません。
- 一般的にプリンタやコーヒーマシンなど、ワークスペース内に共有スペースを設置するとコミュニケーションが改善されます。

- 異なるラボにいる人とは、同じラボの人よりもコミュニケーションが少なくなる傾向があります。
- 国や人種が異なる人同士は、文化的にも言語的にも、コミュニケーションの問題を抱えている可能性があります。
- タイムゾーンが異なる場合、コミュニケーションはより困難になります。

原則として、地理的に分散して働くことは危険ではありませんが、それらの人々が互いにコミュニケーションしなければならない状況が危険が生じます。

たとえば、彼らがシステムの共通部分で作業する場合などです。

あなたは、ソフトウェア構造の中でメンバー間の良好なコミュニケーションを要する領域はどこかを事前にチェックしておく必要があります。これらの領域は地理学的には同じ場所が良いでしょう。

その他考慮される要因としては、

- 新規開発か再利用か：完全に(0から)フルスクラッチで開発された領域は、（主に）再利用される領域よりも多くの欠陥を含む可能性が高くなります。
- インターフェース：多くの欠陥は、しばしばコミュニケーションの問題に起因して、コンポーネント間のインターフェースに関連していることが実証されています。したがって、より多くのインタフェースを持つコンポーネントは、欠陥を起こしやすくなります。この文脈における特徴的な問題は、しばしば、内部インタフェースと外部インタフェースとの間で発生します。
- サイズ：コンポーネントが大きくなり過ぎると、メンバー間で全体間が失われることがあります。したがって、（あまりにもコード量が多い）大きなコンポーネントは、普通のサイズのコンポーネントよりも多くの欠陥を含んでいる可能性があります。

プロジェクトについて知識が少ない場合や、欠陥発生源が特定できない場合は、探索的テストを実施します。

最初にざっとテストして、欠陥のある領域が見つけます。次のテストでは、欠陥の見つかった領域に集中できます。

最初のテストはシステム全体をまんべんなくカバーすべきですが、非常に浅いテストとなります。

代表的なビジネスシナリオといくつかの重要な障害の状況についてのみ実施してください。

その後、最も多く欠陥が発見された領域を特定し、次のテストでこれらの領域を優先させましょう。

次のラウンドは、優先順位をつけた領域に対してより深く、より徹底的なテストをしましょう。

この2フェーズのアプローチは、テストの前に行われた計画と優先順位と併用することで、どんなプロジェクトでも適用できます。

> ハードウェア、産業プロセス、ネットワークなどと直接コミュニケーションするソフトウェアは、ハードウェア障害、データのノイズ、タイミング問題など、比較的外部の影響を受け易いと言われています。

> テスターとして見積もりを減らしたり、テストの実行対象を絞り込んだりしても、どうか怒らないでください。
>
> なぜならばどんなプロジェクトにおいても予算とリソースは常に制約されており、欠陥をゼロにすることはできないため、テストを絞り込む罪悪感と恐怖にさいなまれることは避けることは出来ないはずです。

7.PRISMAプロセス

このセクションでは、PRISMAメソッドを使用して製品のリスクアセスメントを実施する際のプロセスに関して詳細に説明していきます。

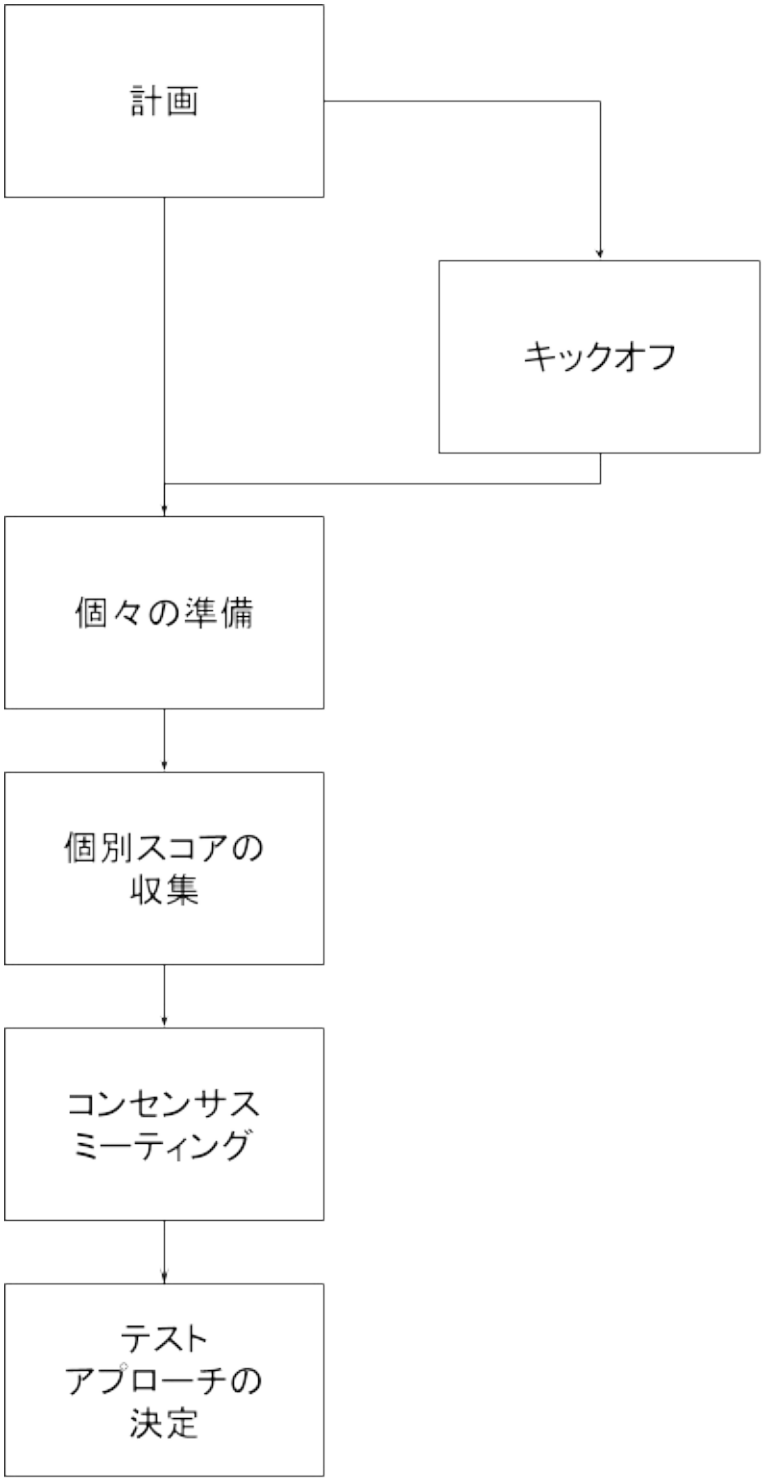

図3：PRISMAプロセスの概要

PRISMAプロセスの中心テーマとしてはいわゆる製品リスクマトリックスの作成です。

前のセクションで説明したように、影響の大きさと欠陥の可能性からテストする各項目（リスク項目）に関する因子が決定されます。

影響の大きさと欠陥の可能性の両方に対して点数付けすることで、リスク項目に関して製品リスクマトリックスを作成できます。

標準的なリスクマトリックスは4つの領域（象限I、II、III、IV）に分かれており、それぞれ異なるリスクレベルとタイプを表現しています。

リスクレベル/タイプが異なれば、それぞれに対して異なるテストアプローチがテスト計画書に記載されるはずです。

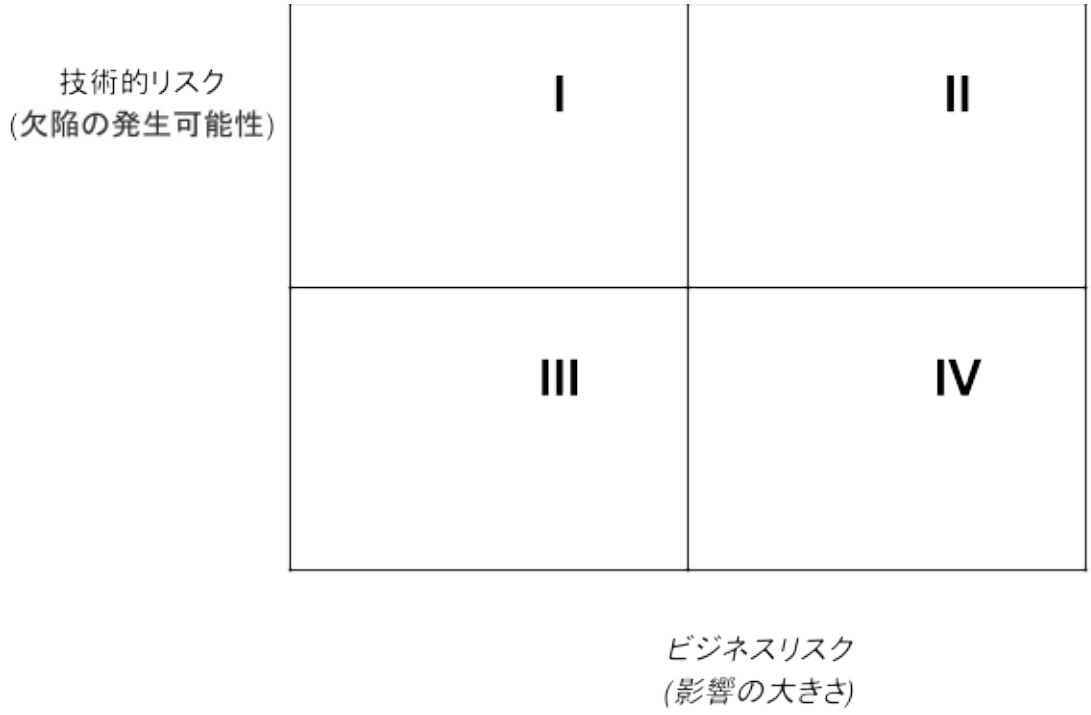

図4：製品リスクマトリックス

PRISMAプロセスをサポートするために、フリーソフトウェアのサポートツールが開発されています。

このツールを使った例は、この本の後半部分で使われています（図5,6,8など）。

もし、すぐに製品の最悪の領域を見つけられたなら、それの領域に対してより多くテストができ、より多くの欠陥を発見できます。

重大な問題が多く見つかった場合、これはしばしばマネージャーにより多くの時間とリソースをテストに割くように動機づける事につながります。

7.1計画

インプットとなるドキュメント（テストベース）の収集

構造された計画の段階は常に成功の鍵となります。

計画中に、インプットとなるドキュメント（テストベースと同じことを指すことが多い）を特定し、収集します。

もちろん、関連するドキュメントはリスク分析対象のテストレベルに大きく依存します。

ドキュメントが要求された品質レベルであることと、このプロセスで使用できる項目（テスト項目と呼ばれる）がちゃんと含まれていることを確認する必要があります。

このフェーズでの テストベースは「最終的である」または「受け入れ済みである」必要はないのですが、製品リスク分析に使用するのに十分安定している必要はあります。

ギャップ（例：要件）を特定し、ドキュメントオーナーに報告する必要があります。

もしテストベースが先述したレベルで利用可能でない場合は、製品リスク分析プロセスを継続する方法を考えたりとそもそもこのプロセスを続行するかどうかを検討する必要があります。

理想的には、ここでいう使用するドキュメントは、要件定義書（システムレベルでのテスト用）またはアーキテクチャを表したドキュメント（開発テスト用）を指します。

リスク項目の特定

リスクアセスメントに使用される項目は、テストベースやリスク項目に基づいて識別されます。

リスクが想定される場合（例えば、ドキュメントのあいまい性に関連する項目等）、これらはドキュメント化されるべきです。

最も確度高くこれらを識別するには、ドキュメントオーナーやステークホルダーとミーティングしてインタービューをする必要があります。

経験則として、プロセスを現実的にすすめるためにはリスク項目はだいたい30〜35項目を超えない位がおすすめです。

つまり、ドキュメントに記載されている項目（要件など）を論理別な単位で分割し、グループ化する必要があります。

識別されたリスク項目は、階層構造で構成されます。

個別のリスク項目として要求ドキュメントに記載されているすべての要件を記載することは、あまり有益はありません。

テスト戦略に応じて、プロジェクトによっては、より高いレベルのリスク分析結果を入力として使用し、後工程でコンポーネントまたはサブシステムごとの基本要件を評価するために、個別のより詳細な製品リスク分析は後回しにすることを決定する場合があります。

識別されたリスク項目リストは、一意に識別され、プロジェクト全員が理解できるものでなければいけません。

各リスク項目は参照IDとして数字で表すことも構いませんが、プロジェクトで意味のあるコードまたは省略形で表現されることもあります。

リスク項目ごとに、可能であれば、テストベースの関連部分にリンクを貼る必要があります。

リスク項目毎に一行ずつ説明を追加する場合もあります。

プロジェクトメンバーにとって、この説明は、リスク項目を評価しなければならないという明確な動機を与える表現であるべきです。

影響度（障害が発生した場合の影響度）因子と尤度（障害が発生する可能性）因子を決定する

製品リスク分析では障害が発生する可能性ともし本番で発生した場合の影響度という2つの成分で分析します。

セクション5と6でこれらの2つの成分について議論したように、影響度や尤度に影響を与えるいくつかの因子を使用することをおすすめします。

テストマネージャーは、プロジェクトが尤度と影響の観点からリスク項目を評価するためにどの因子を使用するかを決定します。

テスト戦略において、リスク因子の標準セットは既に組織レベルで決定されているのが理想的です。

もちろん、プロジェクトによっては、テスト戦略で決めたリスク因子の標準セットから逸脱したものが存在する場合があります。

さらに、各因子は数値化されている必要があります。

さらに、これらの値は単なる数字ではなく'5個未満のインターフェース'、'10個以上のインターフェース'などと意味のある説明で表示されることが好ましいでしょう。

可能であれば、値は客観的かつ測定可能な意味を持つ必要があります（例：'中規模'という表現ではなく<1 KLOCとか）。

しかしながら、現実的な運用上、1,2,3や1から5または0,1,3,5,9などの数値セットが使用されることが多いでしょう。

まず手始めに適用するなら、最も簡単な方法、低（1）、中（2）、高（3）で表現するのも良いでしょう。

純粋にテストの観点だけを考えると、0,1,3,5,9の表現の方がベター です。なぜなら"9"が他の値と比較して明確に区別され、高いリスクであることをより明確に識別できるからです。

1,2,3や0,1,3,5,9など、どの数値セットを使うかは、プロジェクトレベルよりも組織レベル等高いレベルで事前に決定されていることが好ましいでしょう。

各因子の重みを定義する

1つの因子2倍がより重要と考えられる場合、重み付けを使用することも可能です。

例えば、ある因子は、別の因子よりも2倍の「価値」を持つと表現できます。

因子の重み付けは、テスト戦略で既に決定されている事が好ましいが、プロジェクト固有の目的に合わせて調整することもできます。

一般的な方法としては、重み付けを各因子に掛けて、システムの各領域に対して加重和を計算することです。結果的に最も重要性が高い(点数が高い)領域に対して集中的にテストしてください！

選択したすべての因子に対して相対的な重み付けをかけ合わせます。

長く時間をかければ非常に厳密に計算できますが、多くのケースで、重み付けは3つで十分です。

つまり重み付けの値として1,2,3のどれかを使います（1:そんなに重要ではない因子、2:中程度重要な因子、3:影響が強い因子などです。

プロジェクトを幾つか進めていくうちに履歴としてデータが蓄積されたなら、つけた重み付けを微調整できます。

ステークホルダーの選出

製品リスク分析に関与するステークホルダーが特定され、選出されます。

一般的に、ビジネスやプロジェクト内のいろんなロールで選出されます。

例としては、プロジェクトマネージャや開発者、ソフトウェアアーキテクト、マーケティング、エンドユーザ、ビジネスマネージャ、アプリケーションエンジニアなどです。

ステークホルダーの選出は重要なタスクであり、重要なステークホルダーを見逃せば、「関連するリスクが特定されていない」と彼に指摘されてしまいます。

理論的には、ステークホルダーは各々個別にどの因子が重要かという意見を持っていますが、実運用上は、ステークホルダーに関連する、すなわち彼の役割に関連する因子だけを彼にアサインすることが有益でしょう。

典型的には、ビジネス影響を及ぼす因子をビジネス担当者にアサインし、障害の可能性に関する因子は技術の専門家(ソフトウェアアーキテクト、シニアエンジニア等)にアサインすべきでしょう。

明らかに心理的要因が絡んでくるため（自分が関連する因子の重要度を上げたがる、ビジネス担当者はビジネスに関連する因子を重み付けしたくなる)この段階では重み付けを使用しないルールにしましょう。

すなわち各ステークホルダーは同等に重要であると考えましょう。

各因子を少なくとも2人のステークホルダーにアサインすることは特におすすめです。

アサインを担当するテストマネージャは、次の点で良好なバランスを取る必要があります。

- テストレベルに応じて役割に適切な人を選択する。
- 影響度因子は「ビジネスロール」に、尤度因子は「技術ロール」に埋めてもらうようにする。
- 影響度因子と尤度因子の両方について、十分な知識を含んだ上、検討する。

採点ルール

最後にスコアリングプロセスに適用されるルールを設定しましょう。

リスク分析の一般的な落とし穴の1つは、結果がクラスター化してしまう傾向があることです。

つまり、結果は、すべてのリスク項目が互いに近い状態の2次元行列になってしまうことです(すべての因子が最重要など)。

これを効果的に防止するには、ステークホルダーにはすべての因子に対してフルレンジ(1-3や0-9の値のすべてを使う事)でスコアリングするようなルールとするのが良いでしょう。

このルールはPRISMAプロセスをサポートします。

ルールの例としては、「すべての値が使用されなければならない」、「すべての因子が採点されなければいけない」、「空白は許可されない」、「因子に割り当てられた値は均質な分布を取っていないといけない」等です。

> テスト実行の後半部分では、前半で欠陥が発見された領域に焦点を当て、さらに検出された欠陥と同じ種類を対象としたより多く、かつ深いテストを実行する必要があります。

> ドキュメントが要求された品質レベルであることと、このプロセスで使用できる項目（テスト項目と呼ばれる）がちゃんと含まれていることを確認する必要があります。
>
> このフェーズでの テストベースは「最終的である」または「受け入れ済みである」必要はないのですが、製品リスク分析に使用するのに十分安定している必要はあります。

7.2 キックオフ

任意プロセスではありますが、テストマネージャーはキックオフミーティングを開催して、全ステークホルダーにこのプロセスにおけるそれぞれの役割を説明する機会を設けられます。

任意ではあるのですが、キックオフミーティングは強くおすすめします。

なぜなら、キックオフミーティングを行ってしまえば、このプロセスにおいて、すべての参加者それぞれに対して期待される行動は明白になるはずです。

キックオフミーティングは、リスク項目、因子のリストを説明し、どの因子に重点を置く必要があるかを明確にするためにも利用されます。

リスク分析のキックオフ段階では、PRISMAプロセス、リスクベーステストのコンセプト、製品リスクマトリックスだけでなく、この活動の目的も明確化されます。

明確に説明することや共通の見解を作り出すことは、スムーズな運営とステークホルダーそれぞれのモチベーションアップに貢献します。

プロセスの有効性と期待される利益についての説明は、プロセスの後半ではなく、ここのキックオフで展開されるべきです。

ここで共有される項目は次のとおりです。次のプロセスである個々の準備のやり方、使用するツールの説明、タイムスケジュールについて説明します。

テストマネージャーは、キックオフの後の残りのプロセスの概要も共有します。

ステークホルダーには、このフェーズの後半で開催されるコンセンサス会議において何を期待し、さらにプロセスの最後には何が起こるかを明確に説明すべきです。

テストマネージャーは、ステークホルダーに対して、このPRISMAプロセスの役割と貢献がプロジェクトのテストアプローチにどのような影響を与えるかを説明します。

リスク項目と因子については、ステークホルダーに採点を依頼する際に詳細に説明すべきです。

もちろんリスク項目、因子、採点スコアそのスコアそれぞれの正確な意味も、明確にして置く必要があります。

この採点で信頼できる結果を得るには、リスクアセスメントのすべての特性について共通の理解が必要です。

キックオフミーティングの終わりには、ステークホルダーからのコミットメントを得る必要であり、彼らが意欲的に参加することを促す必要があります。

7.3 個々の準備

このフェーズでは、参加者は、リスク項目の各因子に対して値を割り当てます。

それぞれのリスク項目に関して対応する因子の予想されるリスクに対して、最も適した値を選択することによってスコア付けしていきます。

この手順は手動で行うこともできますが、スコアシートの自動チェックをサポートするExcelシートを提供することで、スムーズに行えるでしょう。

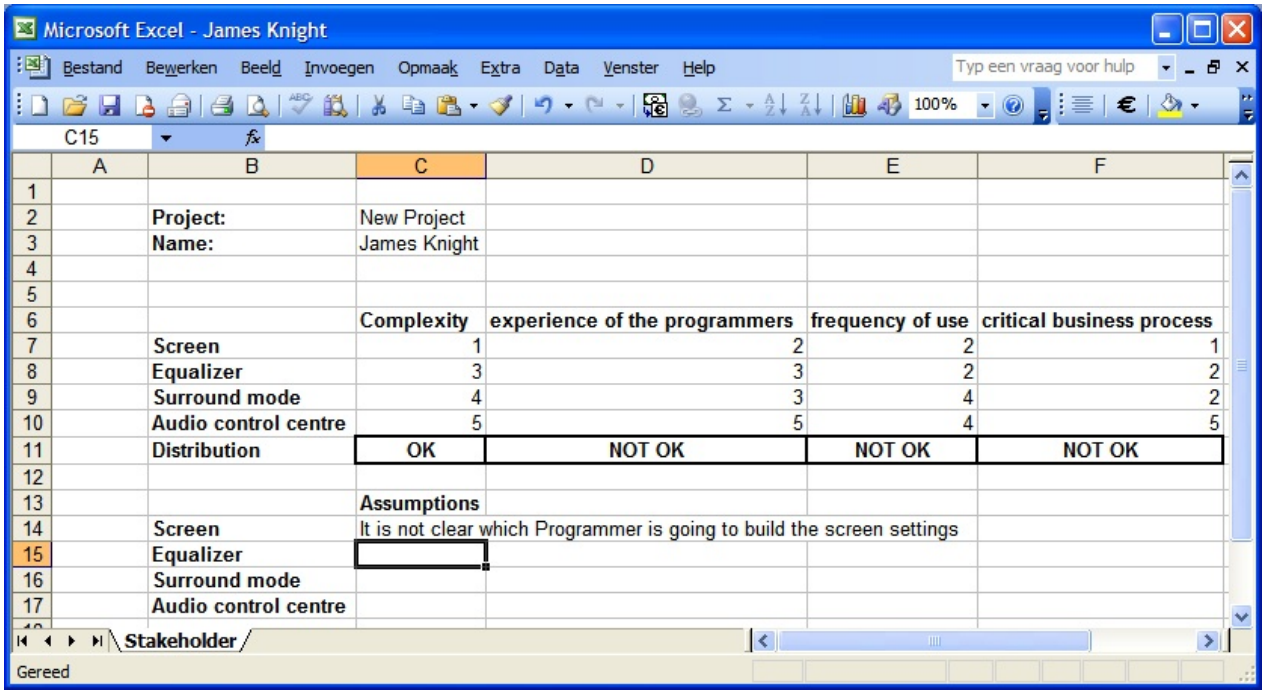

図5：参加者のスコアシートの例（PRISMAツールから）

ステークホルダーが記入した値は、予想されるリスクに基づいています。

つまり各ステークホルダーは何か不具合が生じる可能性がある（欠陥の可能性がある）、もしくは、ビジネスにとって重要なもの（欠陥の影響）を予想して採点しています。

予想されるリスクは、多くの場合、ステークホルダーの個人的な前提に基づいています。

これらの前提もドキュメント化する必要があります。

後で、コンセンサスミーティングにおいて、これらの前提は、スコアの比較、予期しない結果や外れ値を説明するのに非常に役立ちます。

例えば、なぜ私たちはこのようにスコアをつけたのですか？そして、私たちはもしかして異なる前提を持ってましたか？など。

それぞれの値を、テストマネージャが事前に決定したルールと照合していきます。

たとえば、選択した値は十分に分散されており、ステークホルダーにすべてアサイン済みの因子はスコア付けが完了しているかどうかなど。

特定の因子について採点する際には、可能な限り様々な値の間で（できるだけ）均等な分布になることが重要です。

これは、有益な製品リスク分析をするには不可欠です。

値のアサインは絶対値ではなく、相対値です。

例えばどの因子が最も頻繁に使用されるのかとか、最も複雑な因子かとかなど。

スコア処理中に高い値と低い値の両方を使用することにより、より差別化されてスコアリングでき、これは後工程でより明白にテストの優先順位付けができます。

採点の例:

良い例(均等な分布):

リスク項目	複雑度
001	低
002	高
003	中
004	低

悪い例(不均等な分布):

リスク項目	複雑度
001	高
002	高
003	中
004	高

表1：良いスコアリングの例と悪いスコアリングの例

もしこのフェーズを初めて行う場合、テストマネージャは、参加者に対してサポートを行います。プロセス、ルール、場合によっては各因子を詳細に説明します。

7.4 スコアの収集

エントリチェック

このフェーズではテストマネージャーは最初に、個々のスコアの採点中にスコアが正しく付けられたかどうかをチェックします。ルール違反がある場合は、テストマネージャーがこれに関して該当参加者とちゃんと話し合う必要があります。

おそらくこれが発生した場合の解決策として、因子や値の意味を明確にすべきか、もしくはいくつかの追加の前提を作り、ドキュメント化しなければいけないかもしれません。

テストマネージャーは、参加者が前提をドキュメント化しているかどうかも確認する必要があります。

必要に応じて、該当する参加者に対しては前フェーズの個別の準備を（部分的に）やり直してもらう必要があります。

各因子に対して少なくとも2つ以上の異なるスコアがつけられていた場合、テストマネージャーが検査を行う必要があります。

ルール違反をステークホルダーとテストマネージャがどうしても解決できない場合は例として次のような対応をとることになります。

彼らは「スコアの均等分配」が特定の因子に適用できないこと(全部優先度高など)を主張する場合、これはコンセンサスミーティングで議論の対象となります。

スコアの提出期限が近づいているとき、テストマネージャーはステークホルダー達に時間内にスコアを提出するようにリマインドすべきです。

期限内に提出しなかったステークホルダーに対しては、個別にアプローチしてください。

プロセスの次のフェーズに進むには、少なくともプロジェクトのステークホルダーまたは役割にアサインされた各グループからの代表的なスコアの提出がないといけません。

個々のスコアの処理

テストマネージャーは、個々のスコアを計算して平均値を出します。

彼はまた、コンセンサスミーティングで議論されるべき課題リストも準備します。

各リスク項目について、不具合の可能性および影響度が決定されていきます。

つまり、リスクごとの項目で、尤度を決定する因子のスコアが合計され、影響度因子のスコアが合計されます。

各リスク項目は、いわゆるリスクマトリクスに配置できます。

現時点では、コンセンサスミーティングで議論される課題リストの候補はすべて未解決です。

ルールの違反：

- 先述したように、ステークホルダーと一緒にテストマネージャーがルール違反をエスカレーションすることを決定したとき。
- すべての評価フォームの合計結果が未解決のリスク項目につながった場合。因子に対するアサインされたすべての値の分布が所定の閾値を超える場合、リスク項目は「未解決」として判断されます。たとえばあるステークホルダーが特定のリスク項目に関する同じ要素に対して最も高い値を割り当てたのに対して、別のステークホルダーが最小値を割り当てた場合などです。
- また図の6のリスクマトリックス円内のように、すべての象限のボーダーライン上にあるつまりリスクマトリクスの中心に寄りすぎているリスクアイテムも議論の対象となるでしょう。

しきい値およびその他のルールは、計画フェーズ中にテストマネージャが設定したプロジェクトルールに基づいて決定されます。

> レビューされていなくて、実際にプロセスに対して直接インストールされた製品は利用する前にちゃんと出力をレビューされている製品に比べてクリティカルでしょう。
>
> 製品がプロセスを管理する場合、このプロセス自体が分析対象となります。

> リスクの追跡は、プロジェクト全体を通じて行う必要があります。
>
> これは、リスク分析の一部を定期的に繰り返すことによって、また初期リスク分析を繰り返し評価することによって実行できます。

7.5コンセンサスミーティング

コンセンサスミーティングの冒頭でテストマネージャーからミーティングの目的が説明されます。

このミーティングの最後に、（認識される）製品リスクについて共通の理解が達成されるべきです。

最終的な結果は、ステークホルダーがコミットし、ルールセットを遵守するリスクマトリックスで表現されなければなりません。

しかし、全スコアに関して、全員のコンセンサスを取る必要は必ずしもなく、時には不可能な場合もあるでしょう。

結局、各参加者は、自分が抱えている背景と役割に応じて、特定のリスク項目の重要性に関して独自の関心と意見を持っています。

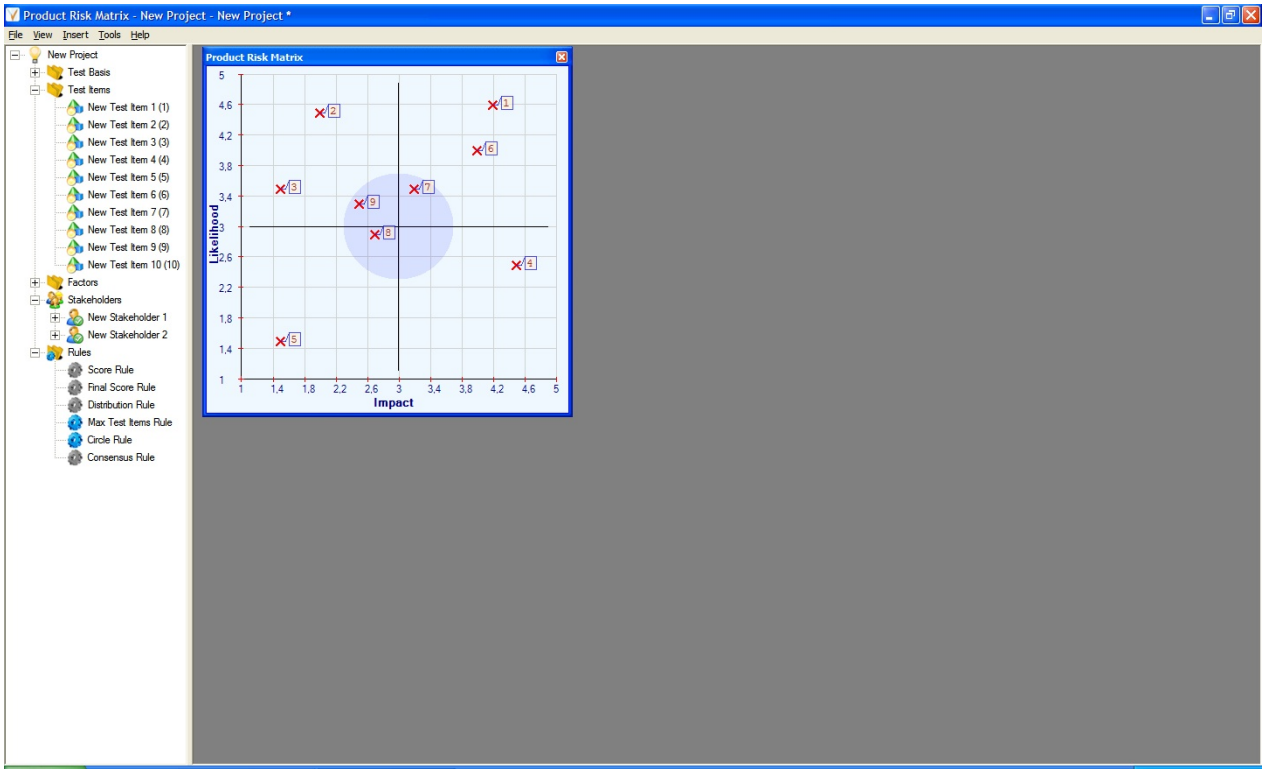

図6：リスクマトリクスの例（PRISMAツールから）

ミーティング中に、未解決リストの項目がステークホルダーの間で議論されます。

これはテストマネージャーが、お互い合意に達し、それぞれの主張を理解することを目的としています。

議論は、ドキュメント化された前提を使用して因子ごと、リスク項目ごとに行われます。

多くの場合、異なるスコアリングは要求の異なる理解に起因しています。

この場合、それぞれの理解が明確化されていないため、要件の変更リクエストにつながります。

ディスカッションの終わりに、最終的なスコアが決定され、結果として製品リスクマトリックスが表示されます（図6参照）。

この結果得られるマトリックスは、ステークホルダーとともに常に検証されるべきです：

「マトリックスが期待どおりのものか、それとも予想を反した結果がありますか？」結果がステークホルダーの期待通りでない場合は、（再）検討する必要があります。

常識は唯一の判断軸ではなく、さまざまな判断方法の一部にすぎません。この常識を使いすぎると危険です。

会議の最後に、テストマネージャは結果を要約し、製品リスクについて共通の理解が達成されているかどうかをチェックします。

必要に応じて、フォローアップミーティングを開催することもあるでしょうし、小規模グループに分かれての特定のディスカッションミーティングを開催することもあるでしょう。（例：要件を担当するチームがミーティング中に質問を提起された場合等。)

大規模プロジェクト：製品リスクマトリックスの拡大

1つのプロジェクト内で、複数の製品リスクマトリックスを作成できます。例としては、

- 受け入れテストのマトリックスとサプライヤーテストのリスクマトリックス。
- マスターテストプランレベルのマトリックスと特定のテストフェーズのマトリックス。

複数のリスクマトリックスが作成されるときは、いつも一貫していなければなりません。

つまり、より低レベルのテスト計画に、より高いレベルのテスト計画からマトリックスの部分だけを再利用することは可能でなければいけません。

例えば、下位レベルのテスト計画ではリスクマトリックスのステークホルダーを選出する場合、ビジネスリスク（インパクトファクター）を評価するステークホルダーを選出しないことがあります。

その代わり、ビジネスリスクに関しては上位レベルのテスト計画からコピーされます。

一貫性を保つとは、複数のマトリックスに存在するリスク項目が同等のスコアを有することを意味しています。

より高いレベルのテスト計画からのリスクマトリックスのリスク項目が、より低いレベルのリスクマトリックスでいくつかの項目に分割される場合、これらの項目の平均リスクスコアは、より高いレベルの製品リスクマトリックスのリスク項目のスコアと同等でなければいけません。

7.6 優先順位付けされたテストアプローチの定義

リスクマトリクス内のテスト項目の位置に基づいて、すべてのテスト項目が優先順位付けが決定されます。

その結果、すべてのテスト項目の実施順序が決定されます。

最初に実施されるべきは最も重要な項目です。

加えて、優先順位付けされて、テスト項目に分解されたテストアプローチは、リスクマトリックスにおけるそれぞれの位置に基づいて定義される必要があります。

通常、テストアプローチは、テストの深さとテストの優先順位の2つの重要な側面を持っています。

テスト深さに応じて、異なるテスト設計手法を用います。

例えば、リスクの高いテスト項目についてはデシジョンテーブルを使用し、リスクの低いテスト項目には同値分割を使用する等の色を付けます。

注意しなければいけないこととしては、テスト深さに応じたテスト設計手法の適用は、すべての（テスト）プロジェクトに対してこの手法を適用できるほど十分に成熟しているわけではないということを念頭においてください。

多くのテストプロジェクトは依然として要件を満たすテストケースのみを作成し、テストデザインを実施していません。

異なるテスト設計手法を使用するという方法以外に、結果のリスクマトリクスに基づいて個別にテストアプローチを定義する代替案があります。

例えば、静的テスト、テストデザインのレビュー、再テスト、回帰テスト、テスト実行の独立のレベルを高める事、およびステートメントのカバレッジ目標などの終了基準などです。

また、リスクの高い因子は、経験を積んだテストエンジニアをアサインするとはリスクを軽減する別の方法です。

ここで言及したテストプラクティスと、それらを使用して差別化テストアプローチを定義する方法について考えてみましょう。

- 静的テスト:識別されたリスクに基づいて、リスクの高い箇所を入念にレビューを行うことなどです。危険性が高いと思われる領域の入念なチェック等。
- テストデザインのレビュー:リスクの高い領域では、テスト設計（またはテストケース）をステークホルダーまたは他のテスターと一緒にレビューすることもできるでしょう。
- 再テスト:再テスト（確認テストとも呼ばれます）では、不具合が修正された後、リスクに応じて全テストを再実行するか、不具合が生じたした手順だけを再実行しするかを選択します。

- 回帰テスト:もちろん、リスクアセスメントの結果を回帰テストにも適用できます。それによって、回帰テストのケースやリソースを最もハイリスクな領域に対して集中的にカバーする用に調整する必要があります。
- テスト実行の独立のレベルを高める:一人のテスターがテストケースとテスト手順を定義し、別のテスタがテスト手順を実行します。その利点としては、独立したテスト実行者は、テストケースとその実行方法にとってより批判的に捉える傾向があり、その結果、より多くの欠陥を見つけられる可能性が高くなります。コンポーネントテストの場合、2人のプログラマーペアを作り、お互いのソフトウェアをテストしたりもします(ペアプログラミング、一人がテストコードを書き、もう一人がそのコードをパスするようにコードを書く)。
- テストの終了基準:それぞれ異なるリスクレベルに対して、それぞれ異なる終了基準(完了基準とも呼ばれる)を、適用できます。リスクカバレッジの高い領域では、要件カバレッジまたはコードカバレッジ基準を厳格に定義する必要があります。

他に終了基準に利用できるメトリクスとしては、実行されたテストケースの割合、未処理の欠陥の数、および欠陥検出率などが含まれます。

製品のリスクアセスメントの結果は、開発プロセスにも影響を与える可能性があることに注意してください。

開発プロセスで選択いかんによっては、残存製品リスク、特に欠陥を発生させる可能性に強く影響を及ぼします。

当初、製品リスクマトリクスの内容は、プロジェクトの初期段階での認識されるリスクに基づいています。

プロジェクト中に、テストマネージャは、変化状況、そこで得られた情報に基づいてマトリックスをアップデートする必要があります。

DDP（欠陥検出率）、（変更された）前提、更新された要件やアーキテクチャのようなこのプロセスの外で観測されるインジケータも含まれます。

プロジェクトの範囲、状況、要件の変更は、しばしばリスクアセスメントプロセスステップのやり直しを必要とします。

したがって、リスクアセスメントプロセスは反復的実行されます。

> 製品の中で一部の機能は毎回使用されているのに対して、一部の機能は数回しか使用されていないということがあるでしょう。
>
> また一部の機能は、たくさんのユーザが利用しているのに対して、一部の機能は少数のユーザしか使ってないということがあるでしょう。
>
> 使用される頻度に応じて機能に優先順位を付けます。

8. PRISMAの実プロジェクトでの有効性

PRISMAを適用したテストプロジェクトと他のプロジェクト、他のリスクベースステストを適用したプロジェクトもしくは、リスクベースステストを全く適用しなかったプロジェクトとの比較は残念ながらできません。

したがって、PRISMAの適用が、本当に利益をもたらすかどうかを判断するために、既に複数プロジェクトにおいてPRISMAを適用した約20社に対してアンケート調査を実施しました。

この調査方法は、テストマネージャー（およびテスター）が、PRISMAが欠陥を検出するタスク(例えばテストやレビュー工程等)に有益であるかそうでないか、どのように認識されているかを調べる方法をとりました。

この方法の背後にある論理的根拠としては、テストマネージャーは、自分達の仕事をより良くするために役立つと信じる確信度に影響して、メソッドを使用するか使用しないかを選択する傾向があるからです。

この決定要因は、PRISMAメソッドの認識された有用性と呼びます。

しかし、テストマネジャは、あるテクニックが有用であると信じても、使用が困難で、より組織的に使用することでパフォーマンスの低下がメリットを上回ると考えるかもしれません。

したがって、有用性に加えて、認識された使いやすさも考慮に入れるべき、第2の重要な決定要因です。

これらの概念をより詳細に定義するために、以下の定義が導入します。

- 認識された有用性とは、特定の方法や技法を使用することで自分の仕事の効率を向上させると信じる確信度です。これは、「有益に使用できる」という用語の定義に従いました。したがって、認識された有効性の高い方法は、テストマネジャが、得られる受益とそれにかかるコストバランスが良好であると確信しています。
- 「使いやすさ」とは、「特定のシステムを利用するストレスがないと信じる確信度」を指します。これは、「容易さ」の定義に従います。利用にストレレスが少ない技術や方法、ツールは、テストマネージャーが採用する可能性が高いでしょう。

有用性	
U1	PRISMAはどの程度テスト管理タスクを実行するのに役立ちますか？
U2	PRISMAを使用することで、テストプロセスがどの程度効率的になりますか？
U3	PRISMAを使用することで、テストプロセスがどの程度効果的になりますか？
使いやすさ	
E1	PRISMAを実プロジェクトに適用するのはどれくらい簡単ですか？

表2：有用性と使いやすさのためのアンケート調査項目

有用性と使いやすさの概念を測定する客観的な尺度はありません。

したがって、主観的尺度は、1（まったく同意しない）から10（強く同意する）までの尺度で測定されます。

このようなスケールを指定するために線形コンポジットという用語を使用できます。

有用性の概念（すなわち、有用性または使いやすさ）の両方はよく対になる項目として認識されています。

つまり、有用性は効率と有効性の2つの側面に分けられます。

これらの各項目について、アンケート参加者は回答します。

参加者は、各自の意見で同意または同意しない、そしてその程度に関して、各項目に回答していきます。

各回答にスコアが割り当てられます。

表2は、実際のアンケートの内容の抜粋です。

> 少人数で熟練したメンバー達で作業するほうが、中程度の能力メンバーが集まった大規模なグループよりもはるかに生産的です。

8.1有用性に対する調査結果

有用性の調査結果を表3に表示しています。

スコアが10段階であることを考慮すると、PBRアプローチが有用であると考えられます。

有用性調査の結果、PRISMAはテストプロジェクトに期待される利益をもたらすことが明らかになりました。

さまざまな項目に対するテストマネージャーの認識としては、PRISMAが欠陥検出タスクの有効性と効率両方に対して向上させることを示しています。

このことは、PRISMAが有用性、効率の両方でポジティブな結果をもたらすという仮説を裏付けるものです。

有用性		有効スコア
U1	有用性	7.5
U2	効率性	7.2
U3	効果	7.0

表3 PRISMAの有用性の結果

有用性に関する参加者からの興味深いコメントには、次のようなものがありました。

- 時間的プレッシャーのかかるプロジェクトにおいて、正しい決定を下すことをサポートします。
- 「リスク」はビジネス用語であるため、ステークホルダーとのコミュニケーションに適しています。
- テスト見積もりや、どのテストアプローチを選択すべきかの基本的な情報を提供します。
- プロジェクト中のテストの監視やコントロールのためのフレームワークを提供します。
- テスト活動(開発テスト中を含む)のための明確なフォーカスエリアを提供します。
- 製品の最も重要な部分がリリース前にテストされることを保証します。
- リスクアセスメントの結果、検出された欠陥の優先度が高まりました。
- リスクに影響を及ぼす要因について学習し、意識を高め、またプロセス改善に利用できる情報を提供しています。

> フェーズ終了判定は、プロジェクトの初期段階において理想的に考えられ制定されますが、これは残念ながら意思決定を容易にするものではありません。

8.2使いやすさに対する調査結果

有用性調査の結果を表4に示します。

残念ながら使いやすさのスコアは有用性のスコアよりも低いことは明らかでした。

実際には、コンサルタントに対してPRISMAのトレーニングやワークショップなどが必要であることがわかりました。

最初に十分なサポートを受けただけで随分テストマネージャーにとってはこのメソッドが使いやすくなります。

また、このメソッドの使いやすさは、テストの成熟度と組織のテスト意識に依存しているようです（後述の特定のコメントも参照してください）。

どちらのステートメントもスコアから導き出せ、特に使いやすさに対するスコアは標準偏差が大きい結果となりました。

PRISMAの経験が豊富なテストマネージャーは、7以上等より高い得点を挙げる傾向があります。

これらの知見に基づくと、テストマネージャーは、PRISMAを導入する際、初期に実用的なサポートを受けていれば、使いやすいと考える傾向があると結論づけられます。

その結果、テストマネージャーがテスト管理の方法でこのPRISMAを採用する可能性が高くなります。

また、このメソッドを適用する前の組織内のステークホルダーにとっては、実質的には、適用がプロセス変更であることに注意してください。

プロセスは簡単ですが適用は簡単ではありません。

使いやすさ		有効スコア
E1	使いやすさ	6.4

表4：PRISMAの使いやすさの結果

使いやすさに関して参加者から寄せられた興味深いコメントには、次のようなものがありました。

- 多かれ少なかれ独立したリスク項目を正しいレベルで定義し、更に約30項目にグループ化分けすることは面倒です。
- （ビジネス）ステークホルダーを特定し、特に最初から関与してもらう場合は困難な場合があります。
- 何人かのステークホルダーにとっては、(マインド・セットの変更も必要でしょうが)、明示的にリスク所有者になってしまいます。

- 明示的な選択(テスト領域を絞る等)をすることは、（ビジネスの）一部のステークホルダーにとっては理解が難しい場合がありました。彼らは常にすべてが「完全に」テストされるべきと考えるからでしょう。
- 因子の解釈は容易ではないので、キックオフと因子（スコアリングルールを含む）の明確な説明をおすすめします。
- 開発の優先順位のテストの優先順位は必ずしも一致するとは限らず、最も重要なリスク項目はプロセスの比較的後工程で提供されます。そのため、リスクに基づいてテストアプローチを選択することは困難です。また、これは参加するテストエンジニアの知識とスキルレベルにも依存しています。
- 最後に、リスクアセスメントの大部分は、プロジェクトの早い段階で認識されるリスクに基づいています。プロジェクトはダイナミックであり、プロジェクト全体でリスクアセスメントも参加者に早い段階で認識されるべきと理解されており、プロジェクトの各マイルストーンでやり直し（一部）が発生します。

最後に、最近とあるPRISMAを導入した企業が、DDPに関して数年間アルファレベルで測定した後、欠陥数の推移公表しました。

テスト実行のリードタイムが短縮されただけでなく、2006年にPRISMAが導入された後、DDP(欠陥検出率)が約10％向上しました。これは、2007年以降のグラフ推移で示されています。

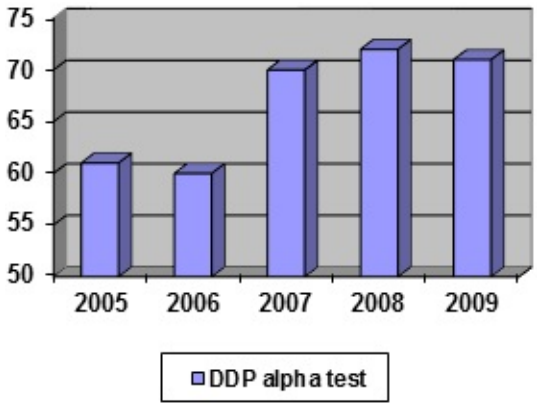

図7：DDP番号アルファテストレベル

> 複雑さというのはおそらく最も注意しなければいけない欠陥発要因です。
>
> 世の中には200種類以上の複雑さを計測する方法が存在し、複雑さと欠陥頻度(欠陥密度)の関連性についての研究は20年以上に渡って続けられています。

> リスクベースのテストでは、特にチームがテストを中断する必要がある場合は常に、利用可能な時間内で最良のテストを行っているという状態を作らなければいけないことに注意してください。

9.リファレンス

• Bach, J. (1997), Good Enough Quality:Beyond the Buzzword, in: IEEE Computer, August 1997, pp. 96-98

• Bach, J. (1998), A framework for good enough testing, in: IEEE Computer Magazine, October 1998

• Boehm, B.W. (1979), Software engineering economics, Prentice-Hall, Englewood Cliffs, NJ

• Jørgensen, M. (1994), Empirical studies of software maintenance, Thesis for the Dr.Sceintific degree, Research Report 188, University of Oslo

• Gerard, P., and N. Thompson, RiskBased E-Business Testing, Artech House Publishers, ISBN 1-58053-314-0

• Karlsson, J. and K. Ryan (1997), A Cost-Value Approach for Prioritizing Requirements, in: IEEE Software, September 1997

• Khoshgoftaar, T.M., E.B. Allan, R. Halstead, G.P. Trio and R. M. Flass (1998), Using Process History to Predict

Software Quality, in: IEEE Computer, April 1998

• Levendel, Y. (1991), Improving Quality with a Manufacturing Process, in: IEEE

Software, March 1991

• Pol, M. R. Teunissen, E. van Veenendaal (2002), Software Testing, A guide to the TMap Approach, Addison Wesley, ISBN 0-201-745712

• Schaefer, H. (2004), Risk Based Testing, in: E. van Veenendaal, The Testing Practitioner – 2nd edition, UTN Publishing, ISBN 90-72194-65-9

• Veenendaal, E. van (2004), The Testing Practitioner – 2nd edition, UTN Publishing, ISBN 90-72194-65-9

10.原著者に関して

原著者であるDrs Erik van Veenendaal（www.erikvanveenendaal.nl）は、

コンサルタント、トレーナー、そして20年以上の実務経験を持つソフトウェアテストと品質管理の分野で広く認知されているスペシャリストあり、Quality Services BV（www.improveqs.nl）の創業者です。

彼はEuroSTAR 1999年、2002年、2005年において、最高のチュートリアルプレゼンテーションを受賞しました。 さらに2007年には長年に渡ってテスト・プロフェッショナルに貢献したことでヨーロッパのテスト・エクセレンス・アワードを受賞しました。

彼は「The Testing Practitioner」、「ISTQB Software Testing Foundations」、「TMapに準拠したテスト」など、数多くの記事や書籍を出版しています。

Erikはまた、アイントホーフェン工科大学の前期パートタイムシニア講師、国際ソフトウェア試験資格審査委員会副委員長（2005-2009）を務めたあと現在はTMMi財団の副議長を務めています。